MATEUS FEIJÓ DE SOUZA

MODELAGEM E VERIFICAÇÃO DE PROGRAMAS DE CLP ESCRITOS EM DIAGRAMA LADDER

FLORIANÓPOLIS
2010

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

MODELAGEM E VERIFICAÇÃO DE PROGRAMAS DE CLP ESCRITOS EM DIAGRAMA LADDER

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia
de Automação e Sistemas.

MATEUS FEIJÓ DE SOUZA

Florianópolis, Outubro de 2010.

MODELAGEM E VERIFICAÇÃO DE PROGRAMAS DE CLP ESCRITOS EM DIAGRAMA LADDER

Mateus Feijó de Souza

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia de Automação e Sistemas, Área de Concentração em *Controle, Automação e Sistemas*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina.'

Jean-Marie Farines, Dr.
Orientador

Max Hering de Queiroz, Dr.
Co-orientador

José Eduardo Ribeiro Cury, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia de Automação e Sistemas

Banca Examinadora:

Jean-Marie Farines, Dr.
Presidente

Max Hering de Queiroz, Dr.

Antonio Eduardo Carrilho da Cunha, Dr.

Leandro Buss Becker, Dr.

Marcelo Ricardo Stemmer, Dr.

Aos meus pais.

AGRADECIMENTOS

Agradeço a Deus, por estar ao meu lado, sempre presente em todos os momentos.

À minha família, por todo o seu carinho, compreensão, incentivo e força, mesmo à distância, indispensáveis para realização deste trabalho.

Aos meus amigos de Pelotas, Porto Alegre e Floripa, pelos momentos sempre agradáveis que passamos juntos.

Aos meus colegas do PPGEAS, pelo companheirismo, pelos grupos de estudo, com muita ajuda nas horas difíceis e momentos de descontração.

Ao DAS e ao governo federal, pelo suporte financeiro fornecido pela bolsa CAPES.

Ao meu co-orientador Professor Max, pela ótima orientação e contribuições sempre relevantes para este trabalho.

Ao meu orientador Professor Jean-Marie, pela excelente orientação, pelos ensinamentos e total apoio para a realização deste trabalho.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

MODELAGEM E VERIFICAÇÃO DE PROGRAMAS DE CLP ESCRITOS EM DIAGRAMA LADDER

Mateus Feijó de Souza

Outubro/2010

Orientador: Jean-Marie Farines, Dr.
Co-orientador: Max Hering de Queiroz, Dr.
Área de Concentração: Controle, Automação e Sistemas.
Palavras-chave: Modelagem de CLPs, Verificação formal, Diagrama Ladder, FIACRE, MDE.
Número de Páginas: xxiv + 162

Para resolver os problemas associados a verificação de sistemas industriais complexos, como os desenvolvidos para CLPs, são necessárias técnicas de modelagem e verificação formal, como forma de provar que o programa está de acordo com as propriedades esperadas. Neste trabalho é proposto um modelo de tradução da linguagem Diagrama Ladder de CLPs para uma linguagem intermediária de verificação FIACRE, que está inserida em uma cadeia de verificação formal do projeto *Topcased*. Esta abordagem segue o paradigma da engenharia dirigida a modelos e consiste em transformar modelos próximos ao usuário em modelos para a verificação. As regras de transformação propostas devem estar inseridas em duas cadeias de verificação formal, que utilizam as abordagens de *model-checking* e por equivalências de modelos. A validação da proposta é feita por intermédio da transformação de modelos e verificação das propriedades de um sistema de automação pneumática e um sistema para um misturador industrial.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Automation and Systems Engineering.

MODELAGEM E VERIFICAÇÃO DE PROGRAMAS DE CLP ESCRITOS EM DIAGRAMA LADDER

Mateus Feijó de Souza

Outubro/2010

Advisor: Jean-Marie Farines, Dr.
Max Hering de Queiroz, Dr.
Area of Concentration: Control, Automation and Systems
Key words: PLCs modeling, Formal verification, Ladder Diagram, FIACRE, MDE.
Number of Pages: xxiv + 162

To solve the problems associated with verification of complex industrial systems, such as those developed for PLCs, it is required formal verification and modeling techniques, as a way to prove that the program complies with the expected specifications. This work proposes a translation model of PLC Ladder Diagram language to FIACRE verification intermadiate language, which is part of a formal verification chain of Topcased project. This approach follows the model driven engineering paradigm in order to translate user models into verification model. The transformation rules proposals must be included in two verification formal chains, using the model-checking and models equivalence. The validation of the proposal is made by models translation and properties verification of a pneumatic automation system and system for an industrial mixer.

Sumário

Lista de Abreviaturas

AADL	Architecture Analysis and Design Language
ATL	Atlas Transformation Language
BCG	Binary Coded Graphs
CADP	Construction and Analysis of Distributed Processes
CLP	Controlador Lógico Programável
CTL	Computation Tree Logic
CTU	Counter Up
EMF	Eclipse Modeling Framework
FBD	Function Block Diagram
FIACRE	Formato Intermediário para Arquiteturas de Componentes e Embarcados Distribuídos
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IL	Instruction List
ISO	International Organization for Standardization
LD	Ladder Diagram
LOTOS	Language of Temporal Ordering

Specification	
LTL	Linear Temporal Logic
LTS	Labeled Transition Systems
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta-Object Facility
POU	Program Organization Unit
SAP	Sistema de Automação Pneumática
SFC	Seqüencial Function Chart
SMV	Symbolic Model Verifier
ST	Structured Text
TINA	Time Petri Net Analyzer
TON	Timer On-delay
TOF	Timer Off-delay
TOPCASED	Toolkit in OPen-source for Critical
Application and SystEms Development	
TPN	Timed Petri Net
TTS	Timed Transition System
UML	Unified Modeling Language
XML	Extensible Markup Language

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Desde o advento dos controladores lógicos programáveis (CLPs), muitas linguagens têm sido utilizadas para escrever programas para máquinas e processos. Os CLPs foram progredindo, ao longo do tempo, a tal ponto que muitos são realmente completos e simulam computadores, executando sistemas operacionais avançados. Os hardwares dentro dos CLPs estão evoluindo constantemente, usufruindo das tecnologias atuais.

Por outro lado, a diversidade das linguagens de programação de CLPs, usadas por diferentes fornecedores, juntamente com sua crescente complexidade, levou a maiores tempos de aprendizado na programação de CLPs. Para este fim, o IEC (*International Electrotechnical Commission*), em 1993, aprovou um conjunto de normas internacionais com a intenção de padronizar a configuração, programação e utilização de controladores industriais. Um dos componentes dessa norma, o IEC 61131-3, define a forma como o usuário pode programar o CLP seguindo a norma, e inclui diversas linguagens de programação [47].

Quando os CLPs começaram a ser utilizados, tendo origem no final da década de 60, um dos requisitos básicos era apresentar uma forma simples de programação, adaptada ao pessoal de campo, responsáveis pela instalação e manutenção. Como as linguagens de programação

convencionais não atendiam este requisito, foi utilizada uma linguagem baseada nos diagramas lógicos de contatos elétricos de relés. Esta linguagem de programação é chamada de Diagrama Ladder (LD - *Ladder Diagram*). Inicialmente, um programa em LD era editado utilizando um terminal de programação, o qual possuía um teclado com símbolos de contato (aberto ou fechado), bobinas e ramificações. Esta forma de programar facilitou o uso do CLP pelos técnicos e engenheiros, não especializados em programação.

O desenvolvimento de programas descritos em LD continua crescendo em termos de quantidade e complexidade [19]. Tradicionalmente, a verificação de programas feitos em LD depende de testes em tempo de execução e simulações baseadas na experiência do projetista. No entanto, simulações e testes podem ser usados apenas para um número limitado de cenários. Como o número de variáveis de entrada pode aumentar, com a possibilidade de ser alterado aleatoriamente, o número de cenários a serem testados cresce rapidamente. A explosão do espaço de estados é uma das principais limitações das abordagens de simulações e testes, junto com a não exaustividade destes. Os métodos de verificação baseados em uma representação formal do programa, tal como a verificação por *model-checking* [14], podem apresentar uma solução interessante para esse problema. A exaustividade deste tipo de abordagem e a existência de ferramentas automáticas de verificação contribuem para diminuir o tempo de desenvolvimento e garantem a correção da programação de CLPs [34].

A crescente complexidade de problemas de controle, podendo envolver vários CLPs em rede, a reutilização de programas existentes e a necessidade de reduzir o tempo de desenvolvimento estão na origem do uso crescente de métodos formais. A demanda por soluções de alta qualidade e, especialmente, as aplicações de CLPs em sistemas críticos, necessitam de procedimentos de verificação e validação formais para provar que propriedades, como por exemplo, de vivacidade, justiça e segurança, serão garantidas [20].

1.1 Motivação

Vários aspectos motivam a realização deste trabalho. De forma mais geral, é de grande importância o estudo dos problemas associados a concepção de sistemas industriais complexos, como os desenvolvidos para CLPs. Este estudo está diretamente relacionado com a qualidade do sistema, em particular, a modelagem e a verificação formal, como formas de garantia de que o resultado final seja o esperado.

De forma mais específica, a motivação deste trabalho está baseada na exigência cada vez maior de qualidade, tanto em programas legados, quanto nos novos. Nestes casos, a existência de um conhecimento dos engenheiros e técnicos na programação de CLPs, tendo como base as cinco linguagens de programação descritas na norma IEC - 61131-3 [26] deve ser considerada. Além disso, a existência de programas industriais de CLPs em operação, que podem ser de grande complexidade, podem também conter erros em certas situações quando são reprogramados ou interligados a outros CLPs.

A ausência da utilização de formalismos, na prática usual, para validar os programas de CLPs existentes, ou os novos programas, escritos nas linguagens da norma, em particular, para a verificação das propriedades, é um fator motivador importante para este trabalho. Existem modelos formais (autômatos, redes de Petri, lógicas) que têm um poder de expressão suficiente para representar os sistemas automatizados e sua programação. Existem ainda, várias ferramentas e ambientes que permitem a verificação das propriedades dos sistemas complexos, como por exemplo, o ambiente Topcased [15] [20] [21] [22] [23] [28]. Além disso, existem linguagens com o poder para expressar aspectos, tanto comportamentais, como temporais dos sistemas, para utilização na verificação, como a linguagem intermediária FIACRE [8].

O projeto Topcased foi concebido para atender o processo de desenvolvimento de sistemas embarcados que integra diferentes tipos de modelos e ferramentas, levando em consideração as características distintas destes modelos e também o longo tempo de vida dos produtos. O

projeto coloca a disposição da comunidade um conjunto de ferramentas de engenharia de sistemas *Open Source* para o desenvolvimento de sistemas críticos de tempo real [42]. O contexto descrito nesta motivação permite validar os objetivos visados neste trabalho.

1.2 Objetivos

O objetivo deste trabalho é propor um modelo de tradução da linguagem LD, de programação de CLPs, para uma linguagem intermediária de verificação, denominada FIACRE [8] (linguagem que está integrada no ambiente de modelagem e verificação em desenvolvimento nas atividades do projeto Topcased [15]). As regras de transformação propostas devem estar inseridas em duas cadeias de verificação formal, que utilizam, respectivamente, as abordagens de *model-checking* e por equivalências de modelos. A modelagem das propriedades a serem verificadas, bem como a validação das propostas e das ferramentas obtidas, utilizando exemplos reais programados em LD, também estão inseridos no contexto deste trabalho.

1.3 Organização do trabalho

A dissertação está dividida em seis capítulos, que serão descritos a seguir. No Capítulo 2 serão apresentados os elementos básicos que compõem os CLPs, desde a descrição da arquitetura básica dos CLPs até as linguagens de programação da norma IEC - 61131-3. Também será apresentado um ambiente de programação de CLPs, utilizado neste trabalho.

No Capítulo 3 será discutida a modelagem e verificação de CLPs, abordando a verificação formal no desenvolvimento de programas para CLPs, juntamente com os trabalhos relacionados com a área. Serão caracterizadas as partes envolvidas na cadeia de verificação formal deste trabalho, que tem como base a engenharia dirigida a modelos. No decorrer deste capítulo, a linguagem intermediária FIACRE e os formalismos

para as ferramentas de verificação também serão apresentados.

No Capítulo 4 será apresentada a transformação de programas em LD para FIACRE. Primeiramente, a modelagem do comportamento de um programa de CLP, descrito em LD, e posteriormente, as regras de tradução de LD para FIACRE e os modelos em FIACRE das construções do LD.

No Capítulo 5 será apresentada a verificação de propriedades para programas de LD. Serão descritas as propriedades para a verificação de programas em LD e a modelagem do ambiente externo, além disso, serão feitas as duas abordagens de verificação, a tradução das propriedades e as aplicações nos exemplos.

Por fim, serão apresentadas as conclusões obtidas com o desenvolvimento deste trabalho, bem como suas contribuições e possíveis trabalhos futuros.

Ainda, no Anexo A, serão descritos os elementos que complementam as linguagens da norma IEC 61131-3. Também serão apresentados, no Apêndice B, os códigos em FIACRE dos programas para CLPs e os resultados completos das ferramentas de verificação.

Capítulo 2

CLPs

A partir do surgimento dos microprocessadores, percebeu-se que eles poderiam fornecer o *hardware* básico para uma forma mais flexível de controle lógico industrial. Os primeiros CLPs (Controladores Lógicos Programáveis) surgiram com a necessidade da empresa *General Motors*, no final da década de 60, devido a grande dificuldade de mudar a lógica de controle dos painéis de comando a cada mudança na linha de montagem, pois tais mudanças implicavam em altos gastos de tempo e dinheiro. Com os CLPs, pôde-se dispensar a utilização dos relés, nas lógicas de comando, que foram substituídos por um software utilizando a lógica *ladder*, dando maior flexibilidade ao comando das aplicações industriais [1] [39].

De acordo com a norma IEC 61131 [26], um CLP é definido como um sistema eletrônico digital, desenvolvido para uso em ambiente industrial, que usa uma memória programável para armazenamento interno de instruções do usuário, para implementação de funções específicas, tais como, lógica, seqüenciamento, temporização, contagem e aritmética, com o objetivo de controlar, em ligação com suas entradas e saídas, vários tipos de máquinas e processos. A Fig. 2.1 ilustra como é fisicamente um CLP, e suas interligações com os elementos externos. A alimentação e a referência, para o CLP e para o conjunto de entradas

e saídas, são representadas, respectivamente, pelas linhas *L1* e *L2*. Na parte esquerda da Fig.2.1 estão representadas as entradas através de chaves, e na parte direita as saídas através de cargas e lâmpadas.

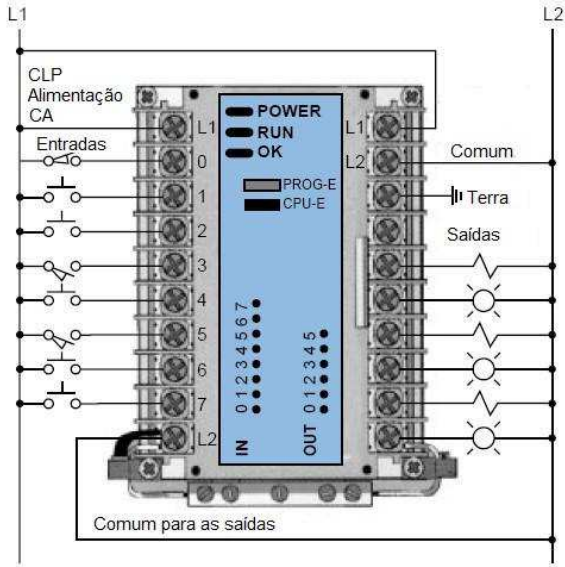


Figura 2.1: CLP e suas interligações com os elementos externos [24].

Um CLP é constituído, basicamente, por uma unidade central de processamento (CPU) e um sistema de entradas e saídas (I/O). A CPU é quem coordena todas as atividades realizadas pelo CLP. Os três componentes principais que formam a CPU são: o processador, o sistema de memória e a fonte de alimentação [1] [39]. A Fig. 2.2 ilustra a estrutura básica de um CLP, onde a interface de entrada permite a conexão entre a CPU do CLP e os dispositivos que fornecem a informação (sinais de entrada gerados, por exemplo, através de chaves) e a interface de saída com os dispositivos controláveis (sinais de saída ligados em cargas do sistema). Durante o seu funcionamento, a CPU realiza três atividades:

- Ler as informações oriundas dos dispositivos de entrada através das interfaces de entrada. Estas informações são armazenadas no

sistema de memória da CPU.

- Executar o programa de controle armazenado em seu sistema de memória. As informações de entradas, necessárias para a execução do programa de controle são lidas do sistema de memória, e não diretamente das interfaces de entrada. Todas as ações de acionamentos de saídas efetuadas pelo programa são escritas também no sistema de memória.

- Acionar os dispositivos de saída através das interfaces de saída. As interfaces de saída são escritas a partir dos valores correspondentes encontrados no sistema de memória.

Estas atividades ocorrem de forma contínua durante o ciclo de execução do CLP, também chamado de ciclo de varredura, mostrado na Fig. 2.3. O tempo de varredura é o tempo total que o CLP leva para completar a execução do programa e a atualização de I/O. Este tempo geralmente depende de dois fatores: a quantidade de memória ocupada pelo programa de controle e os tipos de instruções usadas no programa.

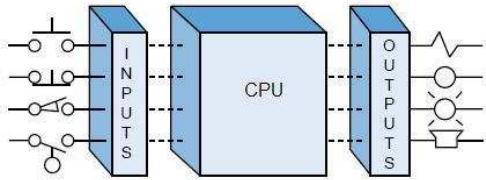


Figura 2.2: Estrutura básica de um CLP [24].

Os CLPs proporcionam muitos benefícios para soluções de controle, desde a confiabilidade até alta capacidade de programação. Além disso, possuem como qualidades: reutilização, reprogramação, maior flexibilidade, maior rapidez na elaboração de projetos e interfaces de comunicação com outros CLPs e computadores [39].

Devido ao grande número de equipamentos e de fabricantes diferentes, é de fundamental importância a padronização das linguagens de programação de CLPs. O resultado da falta de padronização acaba

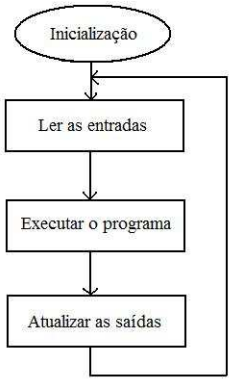


Figura 2.3: Ciclo de execução de um CLP.

ocasionando a necessidade de treinamento em diferentes equipamentos e formação de equipes de manutenção específicas para determinados equipamentos e fabricantes. A conseqüência direta, muitas vezes não percebida pelos programadores e usuários de CLP, é a perda de tempo e recursos financeiros [27] [47]. Para isso, a norma IEC 61131, que é um quadro geral criado em 1993, tenta estabelecer as regras que devem aderir todos os CLPs, englobando mecânica, elétrica e aspectos lógicos. A norma é dividida em cinco partes:

- 61131-1: Informações gerais;
- 61131-2: Requisitos de hardware;
- 61131-3: Linguagens de programação;
- 61131-4: Guia de orientação ao usuário;
- 61131-5: Comunicação.

Outras três partes estão em fase de elaboração:

- 61131-6: Reservada;
- 61131-7: Programação utilizando Lógica Fuzzy;

- 61131-8: Guia para implementação das linguagens.

A terceira parte, IEC 61131-3 [26], que aborda o aspecto da programação de controladores industriais, definindo os blocos lógicos de programação e linguagens de programação, é a parte de interesse deste trabalho.

Conforme apresentado no Capítulo 1, inicialmente os conceitos básicos relacionados ao CLP serão descritos. Neste Capítulo será feita uma caracterização completa dos CLPs, começando por sua arquitetura básica e pelo modelo de software, e serão definidas mais detalhadamente, as linguagens de programação descritas pela norma IEC 61131-3. Por fim, será apresentado o ambiente de programação sugerido pelo órgão responsável pela padronização.

2.1 Arquitetura do CLP

Nesta seção é discutida a arquitetura do CLP, abordando os elementos de *hardware*. Os CLPs são sistemas microprocessados que possuem diversos elementos importantes. Um deles é o processador, que tem como funções principais administrar as atividades de todo CLP, interpretando e executando uma coleção de programas básicos do sistema, chamada de executivo. Ele está armazenado no CLP e, normalmente, não pode ser alterado. O executivo pode também ser considerado como o sistema operacional do CLP. Todas as tarefas de controle interno, processamento de programas do usuário e processamento de comunicações são realizadas com o auxilio do executivo. Também é tarefa dele o controle da comunicação entre o CLP e o usuário, através do dispositivo de programação do CLP, além disso, também pode dar suporte às tarefas de comunicação com outros periféricos.

No CLP, a CPU pode conter mais de um processador para executar todas as tarefas, principalmente àquelas que dizem respeito a comunicação de dados. O multiprocessamento é a divisão de tarefas entre vários processadores, dividindo, por exemplo, o trabalho entre controle e comunicação, podendo aumentar consideravelmente o desempenho de

todo o sistema. Cada módulo contém um microprocessador, memória e um programa executivo pronto para executar somente a função principal.

Outro hardware importante é a fonte de alimentação, no que diz respeito a confiabilidade e integridade do sistema. A sua função não é somente fornecer as tensões de alimentação aos componentes internos do sistema (processador, memória e interfaces de I/O), mas também deve monitorar e regular as tensões fornecidas, e avisar a CPU se algo estiver errado, fornecendo proteção aos outros componentes do sistema.

Um dos componentes mais importantes do CLP é a memória. É a área do CLP onde as seqüências de instruções, ou programas, são armazenadas e executadas pelo processador. Os valores de entrada, saída e os resultados das operações do CLP são também armazenados nesta área. A área de memória, que contém o programa de controle, pode ser alterada, ou reprogramada, para adaptar ao programa de controle as mudanças de procedimentos de linhas de produção, ou para controlar novos sistemas.

O sistema de entradas e saídas (I/O) fornece a conexão física entre a CPU e os dispositivos externos que transmitem e recebem os sinais digitais. Através de diversos circuitos de interface e dispositivos, o controlador monitora e avalia grandezas físicas (proximidade, posição, movimento, nível, temperatura, pressão, corrente, tensão, entre outras) associadas com uma máquina ou processo. Baseado no estado dos dispositivos monitorados ou valores do processo medidos, a CPU envia comandos que controlam os dispositivos no ambiente externo.

Os módulos de entradas analógicas do CLP são usados em aplicações onde os sinais fornecidos pelos dispositivos são contínuos, como por exemplo, a temperatura, que é um sinal analógico. Diferentemente dos sinais digitais, que possuem somente dois estados, os sinais analógicos apresentam um número infinito de estados. As interfaces de entrada analógicas têm ainda a função de converter sinais analógicos contínuos em valores discretos, que podem ser interpretados pelos processadores dos CLPs.

Por fim, outro componente de hardware importante, são os dispositivos de programação. A maioria dos CLPs usa uma forma de programar parecida, utilizando as linguagens de programação, normalmente, definidas pela norma IEC. As principais diferenças na forma de programar os CLPs estão nos mecanismos utilizados para inserir um programa no CLP, estes mecanismos podem variar muito de acordo com o fabricante. Normalmente, cada fabricante disponibiliza aos usuários os dispositivos de programação, que são equipamentos (ou softwares) desenvolvidos especialmente para a tarefa de programar um CLP. Os dispositivos de programação podem ser basicamente: mini-programadores (programadores manuais) ou computadores pessoais [1] [39] [27] [24].

2.2 Modelo de software do CLP

Após a apresentação dos elementos de hardware, que constituem a arquitetura básica, nesta seção são apresentados os conceitos relacionados ao software dos CLPs. A norma IEC 61131-3 [26] descreve um modelo de software proposto. O modelo é composto basicamente por uma configuração, recursos, tarefas e programas. A Fig. 2.4 ilustra uma visão geral deste modelo.

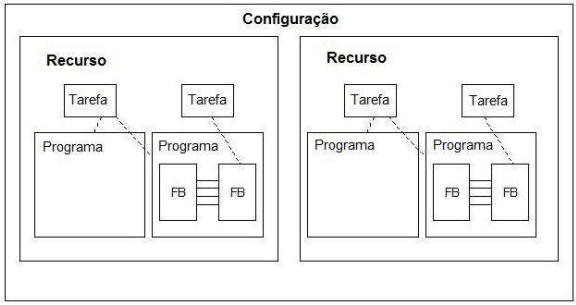


Figura 2.4: Modelo de software [26].

De acordo com o modelo de software, na configuração ocorre a formulação de um programa completo de CLP, requerido para resolver um problema particular de controle. Uma configuração é especificada

para um tipo particular de sistema de controle, incluindo os recursos de *hardware*. Para uma dada configuração pode-se definir um ou mais recursos, permitindo a execução paralela de programas de controle. Os recursos correspondem a uma facilidade de processamento que é capaz de executar programas baseados no padrão IEC. Para um dado recurso, uma ou várias tarefas podem ser definidas. As tarefas controlam a execução de um conjunto de programas e/ou blocos de função. Estas podem ser executadas periodicamente, ou na ocorrência de algum evento. Os programas são construídos a partir de elementos diferentes de software, como funções ou blocos de funções, e escritos em qualquer linguagem definida pela norma IEC.

Para o CLP convencional que contém um recurso e executa uma única tarefa, controlando um único programa, executado em malha fechada, a utilização do padrão IEC 61131-3 oferece mais possibilidades de programação, ou seja, abre novas perspectivas, incluindo multiprocessamento e programas de execução controlados por evento [26].

Há três variações de como programar os CLPs, de acordo com a norma: funções, blocos de função e programa. Essas formas de programação são descritas, com mais detalhes, nas próximas seções.

2.2.1 Funções

As funções têm uma semântica semelhante àquelas utilizadas em linguagens funcionais tradicionais, e retornam diretamente um único valor de saída. No entanto, para além de um ou mais valores de entrada (equivalente aos valores passados como variáveis), a função também pode ter parâmetros utilizados como saídas (o equivalente a passar variáveis como referências), ou como entrada e saída simultaneamente. A apresentação de mais exemplos e a descrição de algumas funções serão tratadas nas seções sobre as linguagens de CLPs. A Fig. 2.5 mostra um exemplo de função, graficamente (utilizado pela linguagem FBD) e textualmente (utilizado pela linguagem ST), que soma os valores das variáveis B, C e D e atribui o resultado na variável A.

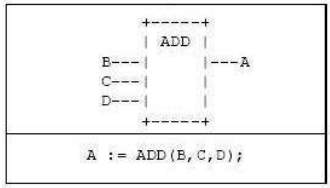


Figura 2.5: Exemplo de função [26].

2.2.2 Blocos de função

Os blocos de função representam um conjunto de funções de controle especializadas. Eles contêm dados e algoritmo, e possuem uma interface bem definida. Permite separar bem os níveis de programação e manutenção. Os blocos de função são instanciados como variáveis, cada um com sua própria cópia do estado do bloco de função. A função padrão de um bloco de função não devolve qualquer valor diretamente, mas pode ter parâmetros para passar dados como entrada, saída ou bidirecional. A Fig. 2.6 mostra um exemplo de bloco de função chamado *Debounce*, que serve para proteção contra trocas de valores dos contatos elétricos, que é constituído por uma interface externa e as funções que compõem o corpo do bloco de funções, de forma textual e gráfica, onde primeiramente são definidos os tipos de variáveis que compõem os blocos, e depois a ligações entre as funções para executar o controle. As funções específicas serão tratadas nas seções sobre as linguagens.

2.2.3 Programas

Os tipos de programas são muito semelhantes aos de blocos de função, com a diferença de que estes só podem ser instanciados dentro de uma configuração, e não dentro de outras funções, blocos de função ou tipos de programa, pois estão em uma hierarquia acima dos demais de acordo com o modelo de software proposto pela norma. Um programa, tipicamente, consiste em um conjunto de funções e blocos de função, que podem trocar dados. As funções e blocos de funções são blocos de construção básicos, contendo uma estrutura de dados e um algoritmo.

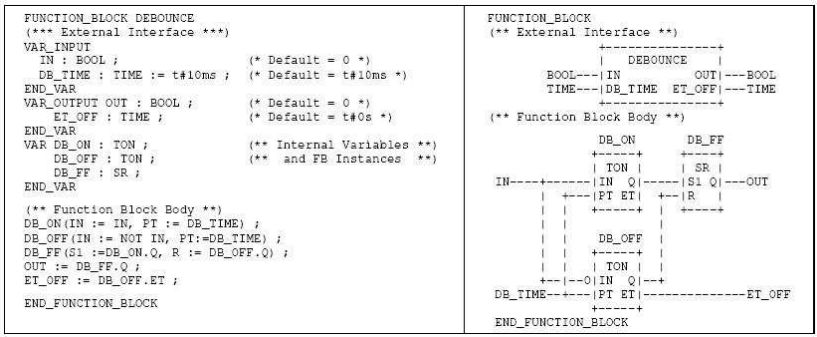


Figura 2.6: Exemplo de bloco de função [26].

A norma IEC 61131-3 [26] define uma grande variedade de funções e blocos funcionais, e todos operam com variáveis. Cada variável, em um programa, deve permitir armazenar um determinado tipo de dado, e todos os tipos de dados são definidos pela norma. Eles estão relacionados com o tipo de informação recebida pelo CLP, tais como informações binárias, números reais, entre outros tipos, dos mais variados de informação. As funções são operações que manipulam estes dados, tais como comparação, inversão e adição. Os blocos funcionais são conjuntos de funções, arranjadas na forma de instruções, criados para trabalhar com blocos de dados. As variáveis globais podem ser usadas por qualquer programa na aplicação, enquanto as variáveis locais podem ser usadas somente por um programa específico.

O padrão IEC permite também que os programadores e fabricantes de CLPs definam seus próprios elementos de programação. Portanto, a norma não define um conjunto fixo e rígido de blocos e elementos para programação, porém estabelece um conjunto mínimo básico de componentes que as linguagens devem seguir, e ainda permite o acréscimo de elementos adicionais conforme as características do CLP e da aplicação em que ele será utilizado. Os fabricantes não são obrigados a seguir a norma, mas a sua adesão, dependendo do grau de conformidade, garante um certificado segundo o padrão IEC [24] [27].

Os três tipos de blocos de programação podem ser programados

em uma das duas linguagens textuais (IL e ST), ou uma das duas linguagens gráficas (LD e FBD), discutidas nas próximas seções. A norma também define uma linguagem gráfica para especificação de máquinas de estado chamada de SFC (*Seqüencial Function Chart*), baseada principalmente em *Grafcet*, que também pode ser usada em blocos de função. Uma vez que uma máquina de estados implica na manutenção do estado, SFCs não podem ser usados para programar essas funções, pois estas são programadas nas outras linguagens da norma e integradas no SFC [26] [47].

Dentre as principais vantagens da norma IEC 61131-3, para a programação de CLPs, é possível destacar a facilidade que o programador tem em estruturar e modularizar a programação de CLPs em elementos funcionais ou unidades de organização de programas, bem como poder definir a linguagem em que irá programar determinada parte do projeto, além de que, aprendendo as linguagens da norma, pode-se usar este conhecimento em diferentes ambientes de programação. Além disso, o modelo de software permite a reutilização de código através da utilização de biblioteca de blocos funcionais, facilitando o desenvolvimento, implantação e manutenção dos sistemas, e aumentando a sua qualidade. Os programas, ou parte deles, poderão ser trocados entre os ambientes de programação através da importação e exportação de módulos [27].

2.3 Linguagens de programação da norma IEC 61131-3

A norma IEC 61131-3 define quatro linguagens de programação de CLPs, e suas sintaxes e semânticas foram definidas eliminando a possibilidade de dialetos. A Fig. 2.7 apresenta um exemplo com as quatro linguagens que descrevem a mesma lógica de programa. As linguagens definidas consistem em duas textuais e duas gráficas:

- Textuais: Lista de Instruções (IL) e Texto Estruturado (ST);
- Gráficas: Diagrama Ladder (LD) e Diagrama de Blocos de Função

(FBD).

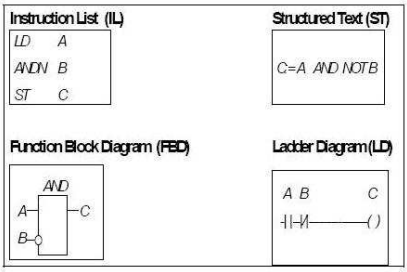


Figura 2.7: Exemplos de linguagens de CLP [19].

A escolha da linguagem de programação de CLP depende:

- da formação do programador;
- do problema a resolver;
- do nível de descrição do problema;
- da estrutura do sistema de controle;
- da interface com outros usuários.

Adicionalmente, a norma inclui uma estrutura que permite a programação similar à Grafcet, este recurso é chamado Diagrama de Funções Seqüenciais (SFC). O SFC é muitas vezes considerado como uma linguagem de programação da norma IEC 61131-3, mas de fato, ele é uma ferramenta que permite criar programas de forma organizada usando as quatro linguagens de programação definidas pela norma.

A norma IEC 61131-3 estabelece um método de programação em blocos gráficos ou orientados a objetos. Este método aumenta a flexibilidade na criação e depuração de programas para os CLPs. O método permite que seções de um programa sejam agrupadas individualmente como tarefas, as quais podem ser facilmente interligadas com o restante do programa. Assim, um programa completo, que siga a norma, pode ser

formado por diversas tarefas pequenas, representando um bloco gráfico dentro de um SFC. Pela norma, é possível a combinação de linguagens durante a criação de um programa de controle no CLP. Este recurso facilita as tarefas de programação e depuração nos CLPs, já que torna possível implementar cada solução de controle através da linguagem de programação mais apropriada.

A seguir será feita uma descrição de cada uma das linguagens de programação definidas pela norma IEC 61131-3. É importante notar que, quando se programa um CLP, que respeite a norma, qualquer uma das linguagens pode ser utilizada, tanto de forma única, como agrupadas. Quando usadas de forma conjunta, normalmente utiliza-se o SFC para estruturar as seqüências lógicas do programa [26].

2.3.1 Lista de instruções (IL)

Como explicado anteriormente, as linguagens de programação de CLP podem ser classificadas de duas formas. Primeiramente, a norma apresenta duas linguagens textuais. Uma delas é a linguagem Lista de Instruções (IL - *Instruction List*). A IL teve sua origem na Europa e se assemelha ao *assembler* (linguagem de máquina). É uma linguagem de baixo nível, e é recomendada para pequenas aplicações e otimizações de código, bem como em aplicações que necessitam de otimização na velocidade de execução do programa ou de alguma rotina específica.

A Fig. 2.8 mostra um exemplo de programa em IL, que faz a operação lógica *and* entre as variáveis *b1, b2 e not b3*. O resultado corrente é mantido em um registrador de resultado e a última instrução armazena o valor do registrador de resultado na variável *b0*. No Anexo A.1 são apresentados os principais operadores e instruções da IL de acordo com a norma IEC 61131-3.

2.3.2 Texto estruturado (ST)

A outra linguagem textual definida pela norma é o Texto Estruturado (ST - *Structured Text*). É uma linguagem de alto nível, que

Instruções		Comentários
LD	b1	resultado corrente:=b1
AND	b2	resultado corrente:=b1 AND b2
ANDN	b3	resultado corrente:=b1 AND b2 AND NOT b3
ST	b0	b0:=resultado corrente

Figura 2.8: Exemplo de um programa em IL.

contém instruções similares as das linguagens Ada, Pascal e C. Esta linguagem contêm todos os elementos básicos de uma linguagem de programação estruturada, incluindo os condicionais (*if-then-else e case of*) e iterações (*for, while e repeat*). O ST é apropriado para a definição de blocos funcionais complexos, os quais podem ser usados em qualquer outra linguagem da norma. Além disso, é ideal para expressar a tomada de decisões, declarar variáveis e configurações, calcular, implementar algoritmos, definir ações, utilizar literais, entre outros [3].

A linguagem ST é extremamente útil para a execução de rotinas como a geração de relatórios, em que as instruções explicam exatamente o que está sendo feito. Deve-se ressaltar que o ST pode ser também usado para encapsular, ou criar, um bloco funcional que realizará uma determinada tarefa disparada por uma lógica de controle. Este bloco funcional pode assim ser utilizado inúmeras vezes por todo o programa de controle. A programação usando a linguagem ST é, particularmente, conveniente nas aplicações que envolvem manipulação de dados, computação com ordenação e com uso intensivo de matemática com valores de ponto flutuante. O ST também é a melhor escolha para a implementação de controles com inteligência artificial, lógica *fuzzy* e controles com tomadas de decisão. No Anexo A.2 são apresentados os principais operadores e instruções do ST [26].

2.3.3 Diagrama de blocos de função (FBD)

A outra forma de classificação são as linguagens gráficas, que da mesma forma como as textuais, possuem duas linguagens descritas pela norma IEC. Uma delas é a linguagem Diagrama de Blocos de Função (FBD - *Function Block Diagram*) que expressa o comportamento de funções, blocos funcionais e programas, como um conjunto de blocos gráficos interligados, da mesma forma que os diagramas de circuitos eletrônicos. O FBD é bastante usado na indústria de processos e tem a característica do fluxo de sinais entre os elementos de processamento. Esta linguagem é adequada para controle discreto, seqüencial e regulatório, e possui uma representação de fácil interpretação, além dos blocos serem expansíveis, de acordo com o número de parâmetros de entrada. Estes blocos são disparados por parâmetros externos, enquanto os algoritmos internos permanecem ocultos [11].

A norma IEC 61131-3 [26] estabelece a forma gráfica de como representar os blocos e as regras para a sua interligação. A norma também estabelece uma determinada quantidade de blocos padrão, que devem ser disponibilizados por todos os sistemas que permitam a programação com FBD. Entretanto, a norma deixa livre aos fabricantes de CLPs disponibilizarem outros tipos de blocos, que possam controlar recursos especiais disponíveis nos CLPs. É permitida também a possibilidade dos usuários criarem seus próprios blocos, de acordo com os requisitos dos programas de controle. A vantagem em criar blocos funcionais é que eles podem ser construídos a partir da ligação de outros blocos, ou utilizando uma das outras linguagens de programação para CLPs, o que cria uma grande flexibilidade na programação por FBD. O encapsulamento permite que um usuário crie novos blocos funcionais, e os armazene em uma biblioteca, podendo ser utilizados diversas vezes no mesmo programa de controle, ou até mesmo em outros programas desenvolvidos por outros programadores. Os blocos personalizados podem ser utilizados em conjunto com os blocos padrão, podendo ser integrados também em programas com a linguagem LD, descrita na próxima seção [26]. A Fig. 2.9 ilustra parte de um exemplo de FBD para um controle

dos sinais através de portas lógicas *and e or*. As lógicas permitem a segurança do sistema entre os sinais para ligar, desligar, permissões e defeitos. No Anexo A.3 são apresentados os principais blocos de função para o FBD.

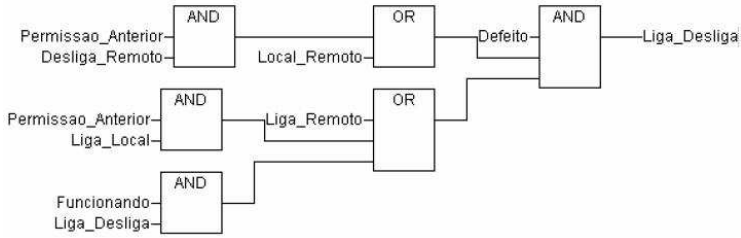


Figura 2.9: Exemplo de um programa em FBD [27].

2.3.4 Diagrama Ladder (LD)

A outra linguagem gráfica definida pela norma é o Diagrama Ladder (LD - *Ladder Diagram*). Esta linguagem teve origem nos EUA e é baseada na representação gráfica da lógica de relés, ou seja, no diagrama elétrico de contatos e foi a primeira linguagem para CLPs. Adequada para expressões combinacionais e sequenciais (intertravamento), ela utiliza ainda blocos de função para controle regulatório e funções especiais. O LD é um conjunto de instruções representadas por símbolos gráficos. Estes símbolos são ligados, de tal forma, a se obter uma lógica de controle, a qual é armazenada na memória de aplicação do CLP. A funcionalidade principal de um programa em LD é de controlar as saídas de um CLP através da análise lógica de suas entradas (lógica *and e or*) e de operações sobre elas. Estes diagramas utilizam passos denominados *rungs* (degraus de escada) para atingir este objetivo. Geralmente um *rung*, onde ocorre o fluxo do sinal, consiste em um conjunto de condições de entrada, representadas por instruções de contatos (*contacts*), ligados em uma instrução de saída no final, esta representada por um símbolo de bobina (*coil*).

A evolução, ao longo dos anos, da linguagem LD original, adi-

cionou a ela um grande conjunto de novas instruções, que podem ser representadas por intermédio de blocos funcionais (por exemplo, temporizadores e contadores). O uso destes blocos funcionais aumenta a flexibilidade e o poder de programação da linguagem LD básica. Quando um programa LD contém um bloco funcional, as instruções de contato são utilizadas para representar as condições de entrada que selecionam ou habilitam o bloco. Um bloco pode ter uma ou mais entradas que controlam sua operação. Além disso, os blocos funcionais podem ter uma ou mais bobinas de saída, que representam o estado da função que está sendo executada pelo bloco [26].

2.3.4.1 Elementos básicos do LD

A Fig. 2.10 apresenta os elementos básicos (contatos e bobinas) que compõem a linguagem LD de acordo com a norma IEC-61131-3, juntamente com suas explicações. A Fig. 2.11 apresenta os contatos e bobinas especiais, sensíveis a borda de subida e descida [26]. A Fig. 2.12 ilustra um exemplo simples de programa em LD, que tem o seguinte comportamento: quando a entrada A é acionada as saídas C *e* D permanecem ligadas, quando B é ativado as saídas são desligadas.

2.3.4.2 Blocos funcionais do LD

Como descrito anteriormente, a linguagem LD, além de contatos e bobinas, possui blocos funcionais, ou simplesmente chamados de funções. Os mais utilizados em LD, e que merecem destaque, são os blocos temporizadores e contadores. O bloco temporizador TON (*Timer on-delay*) consiste em, a partir do momento da ativação da entrada *IN*, ativar a saída Q depois que a contagem do tempo atingir o valor determinado em PT. A saída Q permanecerá ativa enquanto a entrada *IN* permanecer ativa, caindo para zero se a entrada também for para zero antes de atingir PT. A Fig. 2.13 mostra o bloco TON e o seu o comportamento graficamente.

O bloco temporizador TOF (*Timer off-delay*) funciona de forma

diferente, pois consiste em, a partir do momento da ativação da entrada *IN*, ativar a saída *Q* imediatamente, e no momento que desativar a entrada *IN*, contar o tempo determinado em *PT* e desativar a saída *Q*. Se a entrada for para zero antes do tempo determinado em *PT*, a saída permanece ativa. A Fig. 2.14 mostra o bloco TOF e o seu o comportamento graficamente.

O bloco contador CTU (*up-counter*) consiste de, a cada ativação

Contatos estáticos	
Símbolo	Descrição
--| |--	Contato normalmente aberto (NA) O valor ON do lado esquerdo do contato é copiado para o lado direito se o valor associado a variável booleana for ON. Caso contrário, o valor do lado direito do contato é OFF.
--|/|--	Contato normalmente fechado (NF) O valor ON do lado esquerdo do contato é copiado para o lado direito se o valor associado a variável booleana for OFF. Caso contrário, o valor do lado direito do contato é OFF.
Bobinas momentâneas	
--( )--	Bobina (NA) O valor do lado esquerdo da bobina é copiado para o lado direito com o valor associado a variável booleana.
--(/)--	Bobina negada (NF) O valor do lado esquerdo da bobina é copiado para o lado direito. O inverso do valor do lado esquerdo é associado a variável booleana, se o valor for OFF, então o valor associado é ON.
Bobinas com retenção	
--(S)--	Bobina set (latch) O valor booleano associado a variável set é ON quando o valor do lado esquerdo da bobina é ON, e permanece setado até ocorrer a ativação da bobina reset.
--(R)--	Bobina reset (unlatch) O valor booleano associado a variável reset é OFF quando o valor do lado esquerdo da bobina é ON, e permanece resetado até ocorrer a ativação da bobina set.

Figura 2.10: Contatos e bobinas da linguagem LD [26].

Contatos sensíveis a transição	
--|P|--	Contato sensível a borda de subida (P) O valor do lado direito do contato é ON quando ocorre uma transição associada a variável de OFF para ON. O valor do lado direito do contato é OFF em todos os outros momentos.
--|N|--	Contato sensível a borda de descida (N) O valor do lado direito do contato é ON quando ocorre uma transição associada a variável de ON para OFF. O valor do lado direito do contato é OFF em todos os outros momentos.
Bobinas sensíveis a transição	
--(P)--	Bobina sensível a borda de subida (P) O valor associado a variável booleana é ON quando ocorre uma transição de OFF para ON do lado esquerdo da bobina. O valor do lado esquerdo da bobina é sempre copiado para o lado direito.
--(N)--	Bobina sensível a borda de descida (N) O valor associado a variável booleana é ON quando ocorre uma transição de ON para OFF do lado esquerdo da bobina. O valor do lado esquerdo da bobina é sempre copiado para o lado direito.

Figura 2.11: Contatos e bobinas sensíveis a bordas do LD [26].

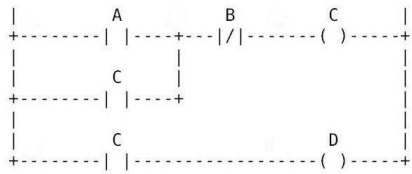


Figura 2.12: Exemplo simples de programa em LD.

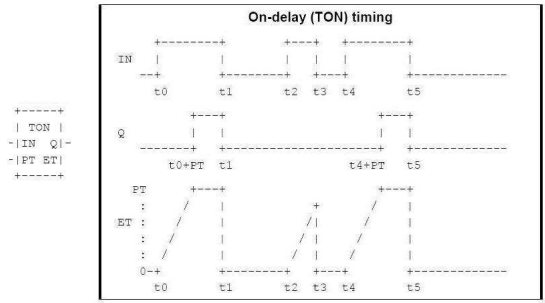


Figura 2.13: Bloco funcional TON e o seu diagrama temporal do comportamento [26].

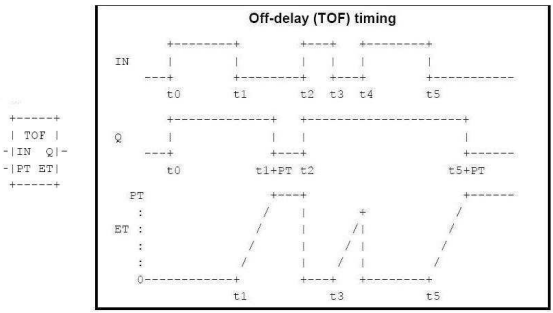


Figura 2.14: Bloco funcional TOF e o seu diagrama temporal do comportamento [26].

da entrada de CTU, incrementar a variável CV. Quando o valor de CV for igual ao determinado na variável PV, então a saída Q é ativada, caso contrário permanece desligada. Antes disso, se a variável R for ativada, zera o contador. A Fig. 2.15 ilustra o bloco funcional CTU, de forma gráfica e o algoritmo da função, de forma textual.

A linguagem LD pode também apresentar saltos (*jumps*) no programa, representados por *labels*. Na varredura do *rung*, quando o *label* é encontrado, o CLP executa o trecho de programa a partir dele e volta para o próximo *rung* da execução normal de onde foi chamado, através da instrução *return*. Os elementos apresentados nesta seção constituem o diagrama Ladder básico e estão descritos pela norma IEC 61131-3

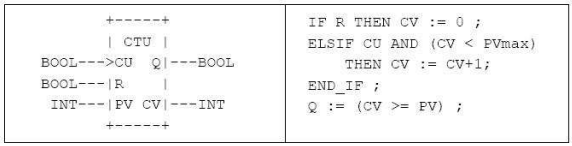


Figura 2.15: Bloco funcional CTU, de forma gráfica e textual [26].

[26]. Cada fabricante de CLP também é livre para acrescentar mais elementos, como blocos funcionais especiais, conforme cada necessidade. No Anexo A.3 são apresentados outros blocos de função para o LD.

2.3.5 Diagrama de funções sequenciais (SFC)

O Diagrama de Funções Sequenciais (SFC - *Sequential Function Chart*) representa graficamente o comportamento sequencial de um programa de CLP. Esta linguagem teve, basicamente, sua origem nas redes de Petri e na linguagem Grafcet, com algumas alterações necessárias para converter a representação de uma documentação padrão para um conjunto de elementos de controle de execução. O SFC organiza a estrutura interna do programa e auxilia a decompor o problema de controle em partes menores, enquanto mantém a sua visão geral.

O SFC consiste em passos, ou etapas, interligados por blocos de ações e transições. Cada etapa representa um estado particular do sistema. Uma transição é associada a uma condição, a qual, quando verdadeira, causa a desativação do passo anterior e a ativação do passo seguinte. Os passos são ligados por blocos de ações, desempenhando uma determinada ação de controle. Cada elemento pode ser programado em qualquer linguagem da norma IEC 61131-3. É possível o uso de sequências alternativas ou paralelas, por exemplo, uma sequência é usada para o processo primário e a segunda para a monitoração das restrições operacionais [3] [26].

Como descrito anteriormente, a linguagem SFC é baseada no Grafcet. No Grafcet, quando uma etapa está ativa, o processador varre a lógica de entrada e saída pertinente às ações da etapa, bem como a

lógica para a transição posterior a etapa. Assim como no Grafcet, o SFC descreve um fluxo de controle e pode ser programado para trabalhar diretamente com temporizações e diagramas de eventos. A maior diferença entre o Grafcet e o SFC é que o Grafcet permite somente comandos de ações escritos, tais como *abrir válvula*, para ligar ou desligar dispositivos. O SFC permite implementar ações de diversas formas, usando as linguagens IL, ST, FBD e LD, ou uma combinação entre elas.

Os SFCs podem ser interpretados como objetos de construção de blocos, usados para criar a estrutura básica de todo o programa de controle, enquanto as outras linguagens podem ser usadas para implementar os detalhes dentro do SFC. Os SFCs podem ter as macro-etapas, as quais permitem um SFC principal ter outro SFC como ação de uma etapa [26]. A Fig. 2.16 ilustra um exemplo de SFC de comando de um sistema de produção onde, a partir do passo inicial, ocorre uma transição para o passo *carregamento de matéria-prima*, que executa uma função em LD. A partir disso, através de suas transições, pode ocorrer o passo *carregamento de lixívia*, ou *geração de vapor*, que executa blocos de função.

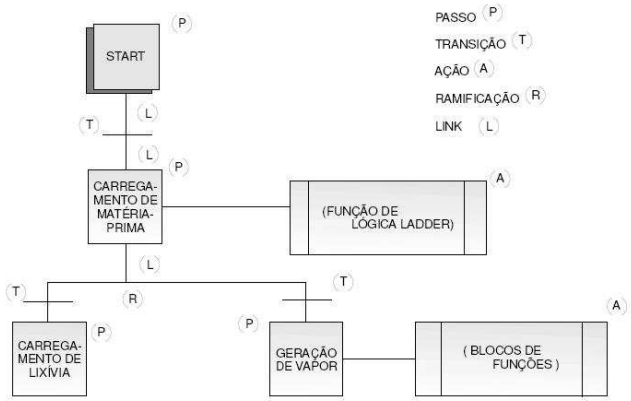


Figura 2.16: Exemplo de aplicação do SFC [27].

2.4 Um ambiente de programação de CLP

Para editar os programas para CLP, a PLCOpen, promotora da norma IEC 61131, desenvolve e mantém o ambiente de programação PLCOpen Editor, que faz parte do conjunto de ferramentas do ambiente Beremiz. Uma das principais atividades da PLCopen está focada na divulgação do padrão IEC 61131-3, a única norma mundial para programação de controladores industriais. A norma harmoniza a forma como as pessoas projetam e operam os controles industriais, padronizando a interface de programação. Uma interface de programação padronizada permite que indivíduos com diferentes formações e habilidades possam criar diferentes elementos de um programa em diferentes estágios do ciclo de vida do *software*: especificação, projeto, implementação, teste, implantação e manutenção.

A PLCopen é uma organização ativa no controle industrial, e está criando uma eficiência para o desenvolvimento do *software* aplicativo: em um único projeto, assim como em um alto volume de produtos. É embasada em ferramentas livres (código aberto) para CLPs, para as quais extensões estão sendo definidas, como bibliotecas para segurança, especificação XML, níveis de reutilização e conformidade. A organização faz sólidas contribuições para a comunidade, estendendo a independência entre o *software* e o *hardware*, assim como a reutilização de código e integração com ferramentas externas de *software*. A PLCopen vem desenvolvendo um formato de troca padronizado, baseado em XML, mas sua utilização não é obrigatória. Os sistemas que entendem esse formato podem ainda receber o certificado de reusabilidade que indica que seus programas podem ser reutilizados em outros sistemas. A compra de um CLP compatível com a IEC 61131-3 exige uma série de critérios. Como não é possível exigir de nenhum fabricante o cumprimento integral da norma, é necessário definir com clareza qual subconjunto da norma é utilizado. Uma forma de garantir o atendimento mínimo às características é verificar se o produto possui a certificação PLCOpen correspondente [40].

O projeto Beremiz [47], mantido pela PLCOpen, contém um ambiente de desenvolvimento integrado, com o editor para as linguagens da norma IEC, um compilador para a linguagem C, comunicação com interfaces para os dispositivos I/O e interfaces gráficas para interação homem-máquina. A Fig. 2.17 ilustra uma visão geral graficamente da composição do ambiente Beremiz, com os editores para CLP, o compilador C e a saída para os hardwares.

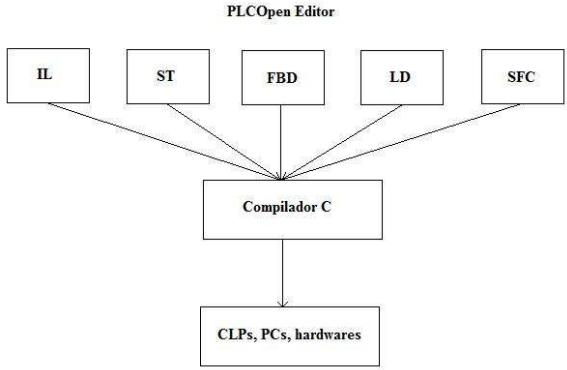


Figura 2.17: Visão geral do projeto Beremiz.

O editor gráfico PLCOpen Editor é um editor das cinco linguagens de CLPs descritas pela norma, seguindo à especificação definida pela PLCOpen. Além de estar totalmente de acordo com a norma, o editor é uma ferramenta livre, gerando como saída arquivos padronizados e com a possibilidade de integração com as demais ferramentas do projeto Beremiz. A especificação utilizada pela ferramenta define uma gramática XML que descreve as linguagens da norma IEC 61131-3. Todos os programas escritos neste ambiente geram arquivos XML, de acordo com esta gramática. É então possível a troca com outros projetos de editores IEC que estejam de acordo com o padrão PLCOpen.

Os editores gráficos SFC, FBD e LD permitem ao usuário inserir e apagar graficamente elementos de programação, de forma a não permitir ao usuário a introdução ilegal no *layout* do projeto. Estes programas são, portanto, sempre corretos sintaticamente, embora possam estar in-

completos. Por exemplo, o editor SFC (Fig. 2.18) prevê um conjunto de ferramentas, com as quais, inserem passos, transições e blocos com ações. A inserção destes elementos (exceto o passo inicial) deve ser referenciada anteriormente para um elemento do SFC. Por exemplo, para inserir uma transição, o usuário deve selecionar o primeiro passo para a qual ele será associado. Da mesma forma, para inserir um primeiro passo, o usuário deve selecionar uma etapa de transição. No caso de um passo ser selecionado, o editor insere automaticamente uma transição entre as fases [47].

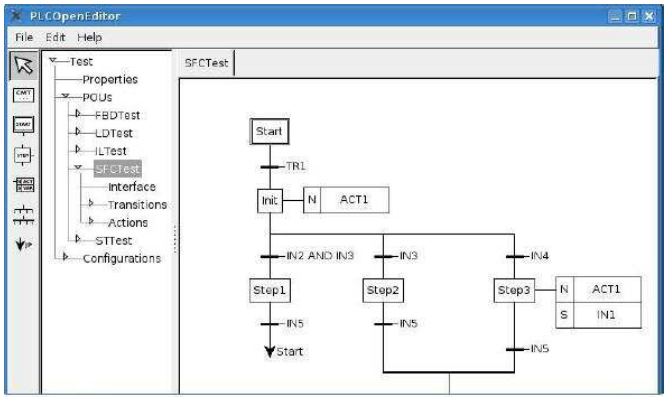


Figura 2.18: Exemplo de utilização do editor SFC [47].

O editor LD segue a mesma filosofia. A criação de um novo patamar implica a inserção de uma saída de relé (bobina). No *rung*, só é permitido ao usuário introduzir novos elementos, de tal forma a produzir outro estado válido. Já os editores de linguagem textual, IL e ST, incluem o destaque da sintaxe do código, e auto-conclusão de palavras-chave e variáveis de nomes. Os erros simples de sintaxe são destacados, no entanto (e ao contrário dos editores gráficos), o código pode ser salvo com a sintaxe e/ou erros semânticos de construções.

O usuário interage com o ambiente IEC através de uma interface gráfica. Esta interface permite que o usuário crie um projeto constituído por várias unidades de organização de programas. Estes estão listados

em uma árvore de exibição do lado esquerdo do painel do IDE (*Integrated Development Environment*). Cada POU (unidade de organização de programa), na árvore, pode ser expandido para mostrar suas variáveis de interface, bem como todas as variáveis internas. Os POUs podem ainda ser programados em qualquer uma das linguagens IEC 61131-3, utilizando uma linguagem adequada no editor que fica no painel do lado direito do IDE [47].

A ferramenta possui um módulo que é responsável por traduzir de linguagem gráfica (FBD e LD) em ST. Esta parte está integrada no editor gráfico, mas pode ser utilizado independentemente. A conversão é também condicionada pelo modo *debug* opcional, que acrescenta as informações necessárias no código gerado. Esta informação será então utilizada em tempo de execução para garantir o *feedback* aos usuários.

O PLCOpen Editor gera como saída arquivos no formato XML. Todos os programas (em uma das linguagens descritas pela norma IEC 61131-3) editados no ambiente de programação são transformados em arquivos com a extensão *.xml*. Esses arquivos podem ser analisados através de um navegador de internet, ou pela própria ferramenta PLCOpen Editor da mesma forma que foi editado. O código XML é organizado de acordo com o esquema mostrado na Fig. 2.19.

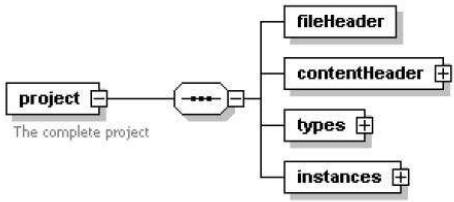


Figura 2.19: Estrutura do código XML [48].

Na parte de *fileHeader* (cabeçalho do arquivo) e *contentHeader* (conteúdo do cabeçalho) estão as informações sobre a criação do arquivo, como nome do projeto, nome do criador do projeto, versão, data, descrição, entre outros. Ainda na parte de cabeçalho ficam as coorde-

nadas (posição na tela) e escalas dos blocos e elementos que compõe as funções e programas do projeto. Em *types* (tipos) são especificados os tipos de dados usados no projeto de acordo com a unidade de organização do programa e todos os detalhes das variáveis utilizadas. Em *instances* (ocorrências) estão as configurações e as utilizações de recursos e tarefas [48].

No arquivo XML, gerado pelo editor LD, na descrição dos elementos, está identificado o nome do elemento que está sendo utilizado. Basicamente existem duas formas de disposição dos elementos em LD, que caracteriza a lógica entre eles. Eles podem estar relacionados de forma *and* (em série) e/ou *or* (em paralelo). No arquivo XML, esta relação entre os elementos é feita através da posição em que eles se encontram. Isto é feito através do elemento da esquerda como referência, ou seja, cada elemento tem como identificação o elemento que está posicionado na sua esquerda, pois segue a lógica de varredura do *rung* (de cima para baixo, da esquerda para a direita). Quando dois elementos têm a mesma referência e ambos são a mesma referência para outro elemento, então estão ligados de forma *or*. Quando uma das condições anteriores não é satisfeita, então ele está ligado de forma *and*.

A Fig. 2.20 representa um exemplo simples de programa editado no PLCOpen Editor e a Fig. 2.21 a correspondência do mesmo exemplo em XML. De acordo com a explicação anterior, no código XML, os elementos são identificados através da variável *contact localId*, a variável *negated* identifica que é um elemento NF (normalmente fechado, ou seja, ativado com valor zero) e possui como referência (onde está ligado) a variável *connection refLocalId*. As outras informações de posicionamento e tamanho do elemento podem ser irrelevantes de acordo com a aplicação.

2.5 Conclusão do capítulo

Foi apresentada neste capítulo a importância da padronização das linguagens de CLP, por intermédio da norma IEC 61131-3. Foi também

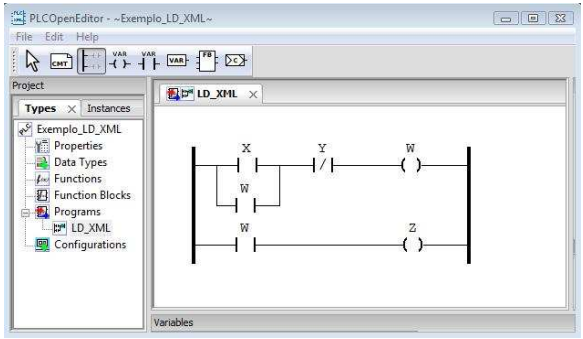


Figura 2.20: Exemplo de programa no Editor LD.

apresentada uma visão geral do CLP, com a arquitetura, modelo de *software* e suas linguagens de programação, que pertencem a norma IEC 61131-3 (IL, ST, FBD, LD e SFC), bem como algumas vantagens da utilização da norma. Finalizando o capítulo, foi descrito o ambiente de programação PLCOpen Editor, pertencente ao ambiente Beremiz da PLCOpen, que promove a norma IEC e que será integrado na cadeia de verificação desenvolvida neste trabalho.

A linguagem de entrada das cadeias de verificação deste trabalho é a linguagem LD, que pertence a norma IEC. Para este trabalho, foi escolhida apenas uma das linguagens, por ser a mais antiga e amplamente utilizada para programar CLPs. Para trabalhos futuros, existe a previsão de que as outras linguagens da norma façam parte do projeto.

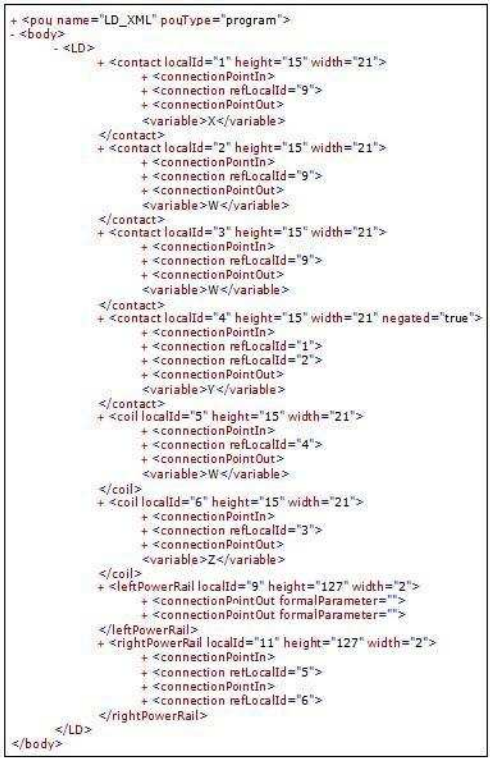


Figura 2.21: Trecho do código em XML correspondente ao exemplo anterior.

Capítulo 3

Modelagem e verificação formal de programas para CLPs

No capítulo anterior foram discutidos os aspectos relacionados com a estrutura, linguagens e ambiente de programação dos CLPs, de acordo com a norma IEC 61131-3. Neste capítulo será tratada a modelagem e verificação formal de programas para CLPs. Será discutida a verificação formal de sistemas, juntamente com as abordagens utilizadas neste trabalho, bem como, os trabalhos relacionados com o uso de verificação em CLPs. Além disso, será discutida a proposta deste trabalho, que se baseia na transformação de modelos da MDE (Engenharia dirigida a modelos). Também serão apresentadas as cadeias de verificação de CLPs, adotadas neste trabalho, e os elementos necessários para o entendimento dessa cadeia, como a linguagem intermediária FIACRE e os formalismos e ferramentas de verificação.

3.1 Verificação formal no desenvolvimento de programas para CLPs

Inicialmente, é importante definir o escopo deste trabalho, que é a verificação de propriedades de programas de CLP, escritos em LD. A validação de programas escritos em LD, e nas outras linguagens de CLP em geral, depende de testes manuais e simulações baseadas na experiência do projetista. Os simuladores podem ser usados apenas para um número limitado de cenários. Entretanto, como o número de variáveis de entrada pode aumentar, podendo estas serem alteradas aleatoriamente, o número de cenários a serem testados cresce rapidamente. A explosão de estados é uma das principais limitações das abordagens tradicionais de verificação.

A verificação formal é o ato de provar se um algoritmo ou sistema está correto de acordo com um conjunto especificado de propriedades. Para isso, é necessária a especificação formal dos requisitos, ou propriedades previstas para o sistema, a modelagem formal do comportamento operacional do sistema e um conjunto de regras que permitem decidir se o sistema está conforme as propriedades esperadas. Neste trabalho, são utilizadas duas técnicas de verificação: *model-checking* e equivalência. Para o *model-checking* é utilizada a lógica LTL para especificar as propriedades do sistema, uma tradução em FIACRE e TTS (sistema de transição temporizado) do programa em LD e a ferramenta TINA para verificar as propriedades em LTL sobre o sistema. Para a equivalência, é utilizada a tradução do programa em LD do CLP em FIACRE e LOTOS, e as especificações traduzidas para a linguagem LOTOS, para ser utilizado na ferramenta CADP. Este trabalho tem como base o desenvolvimento de cadeias de verificação automática de propriedades comportamentais de programas descritos em LD. As propriedades a serem verificadas sobre os programas em LD podem ser genéricas ou específicas, isto é, relacionadas ao modelo [20] [34].

As propriedades genéricas podem ser aplicadas a todos os modelos de programas de LD, e podem ser representadas por intermédio

de propriedades como a acessibilidade, limitação, ausência de *deadlock*, vivacidade, sem ambigüidades e questões de segurança. Outra propriedade genérica que pode ser verificada em programas feitos em LD é a ausência de condições de corrida (*race conditions*). Uma condição de corrida é uma situação indesejável que ocorre quando um dispositivo ou sistema tenta executar duas ou mais operações ao mesmo tempo, porque de acordo com a natureza do sistema, as operações devem ser feitas na sequência adequada para o correto funcionamento. A condição de corrida ocorre quando, a partir de valores de entrada fixos, uma ou mais saídas ficam alternando de valores constantemente [6].

As propriedades relacionadas ao modelo são específicas ao sistema a ser verificado, e são normalmente especificadas pelas empresas por intermédio dos engenheiros que definem as especificações (propriedades) desejadas. Elas estão diretamente relacionadas com a especificação do sistema de controle. Exemplos de propriedades específicas relacionadas ao modelo de um sistema de elevação de caixas de uma esteira, por exemplo, podem ser:

- Qualquer caixa posicionada sobre a mesa de elevação será elevada;
- A mesa de elevação não irá subir sem uma caixa sobre ela;
- Cada caixa elevada deve ser expulsa na parte superior do transportador.

Estas propriedades precisam ser geradas semi-automaticamente (com a participação do projetista). Entretanto, em várias áreas de atuação, são utilizadas formas de padronização para descrever as propriedades, nestes casos, uma tradução destas propriedades é possível de ser realizada. Neste trabalho é exemplificado este caso por intermédio de um exemplo pelo domínio da pneumática. As próximas subseções irão tratar das duas abordagens de verificação formal (*model-checking* e equivalências) que serão aplicadas em programas de LD utilizados neste trabalho.

3.1.1 Abordagem lógica (*model-checking*)

O *model-checking* (verificação de modelos) é uma técnica automática para verificação formal de sistemas de estados finitos. Para fazer a verificação é necessário seguir algumas etapas: primeiramente, é a escrita do sistema em um modelo formal (geralmente um sistema de transição) que o verificador possa interpretar. Também é necessário que as propriedades esperadas para o sistema sejam igualmente formalizadas sobre a forma de uma fórmula de lógica temporal. De posse destas duas representações, ou seja, do sistema e das propriedades, o verificador então faz a checagem das propriedades do sistema, através da satisfação das fórmulas de lógica temporal, representando as propriedades, sobre o modelo do sistema descrito como um sistema de transição. Como resultado pode-se gerar o sucesso, se as fórmulas são verificadas como verdadeiras, ou um contra-exemplo, em caso contrário, apresentando um motivo da não verificação. Duas vantagens que podem ser observadas em relação a métodos tradicionais são: o processo pode ser totalmente automatizado, não exigindo conhecimentos especiais em lógicas ou provas por teorema, e um contra-exemplo é gerado quando uma determinada propriedade é falsa.

A principal desvantagem do método *model-checking* é a situação de explosão de estados que pode ocorrer quando o sistema tem muitos componentes que podem ter transições ocorrendo em paralelo. A verificação de modelos pode ser aplicada em sistemas reativos, que se caracterizam por uma interação contínua com o ambiente no qual estão inseridos. Os sistemas desta natureza tipicamente recebem estímulos do ambiente e quase que instantaneamente reagem às entradas recebidas. Para a verificação, é necessário gerar uma resposta do ambiente e de seu comportamento. Tradicionalmente, eles são complexos, distribuídos, concorrentes e não possuem um término de execução, isto é, eles estão constantemente prontos para interagir com o usuário ou outros sistemas. Este conjunto de características exige que as propriedades destes sistemas sejam definidas não apenas em função de valores de entrada e saída, mas também em relação à ordem em que os eventos

ocorrem [37] [44].

As lógicas temporais são utilizadas para a especificação de propriedades em verificação de modelos, pois elas são capazes de expressar relações de ordem, sem recorrer à noção explícita de tempo. A lógica temporal usa proposições atômicas para fazer afirmações sobre os estados, em que estas proposições são relações elementares, as quais, em um dado estado, possuem um valor verdadeiro bem definido. As duas formalizações de lógicas temporais mais utilizadas, no contexto de verificação de modelos são: LTL (*Linear Temporal Logic*) e CTL (*Computation Tree Logic*). Estas duas lógicas temporais podem ser vistas como um fragmento da CTL* (*full* CTL). A lógica temporal CTL é um subconjunto da lógica CTL*, em que um operador temporal pode ser imediatamente precedido por um quantificador de caminho [25]. Uma fórmula proposicional é uma combinação de proposições e combinadores booleanos (constantes, negação, conjunção, disjunção, implicação e dupla implicação). Na lógica temporal LTL, os operadores temporais permitem construir expressões relacionadas ao sequenciamento dos estados ao longo de uma execução e não apenas aos estados individualmente. Alguns desses combinadores temporais para a lógica LTL, por exemplo, são:

- *Next:* $X\ p$ (quando p é válida no próximo estado);
- *Future:* $F\ p$ (quando p é eventualmente válida no futuro);
- *Globally:* $G\ p$ (quando p é sempre válida);
- *Until:* $p\ U\ q$ (quando p é válida até que q o seja).

Neste trabalho a lógica utilizada é a LTL, pois a versão da ferramenta TINA utilizada neste trabalho, inserida no contexto do ambiente Topcased, possui como entrada um subconjunto da lógica LTL para a verificação formal.

3.1.2 Abordagem comportamental (equivalências)

A equivalência observacional (ou bisimilaridade fraca) é uma noção natural de equivalência, em que dois processos são observacionalmente equivalentes, dependendo do tipo de equivalência que se deseja. A verificação de equivalência observacional é muito utilizada na prática, para mostrar que um determinado processo é equivalente à sua especificação [36]. A verificação consiste em comparar um programa com as especificações, de alguns aspectos ou da totalidade das mesmas.

Como dito anteriormente, a técnica de verificação por equivalência usa a noção de bisimulação fraca. A bisimulação é uma relação binária entre estados de dois sistemas de transições que define a equivalência entre estes, no caso deste trabalho, inicialmente, utilizando a descrição na linguagem FIACRE, e posteriormente, em LOTOS [10]. A primeira etapa consiste em distinguir os sinais que passam a ser considerados como eventos internos. A segunda etapa consiste em reduzir o modelo na sua forma mínima, do ponto de vista da bisimulação. Este modelo reduzido contêm poucos estados e transições, então as propriedades podem ser facilmente verificadas por simples leitura, ou por equivalência com o modelo das especificações.

Por outro lado, a técnica de verificação por observadores consiste em construir um observador, por exemplo, chamado *'O'*, que expressa a propriedade a ser verificada e a colocá-la em paralelismo síncrono com o programa, por exemplo, chamado *'P'*, a verificar. Como o papel do observador *'O'* é de ver o comportamento do programa *'P'*, ele tem ainda como entrada, as saídas do programa *'P'*. Técnicas de analise de alcançabilidade para autômatos permitem verificar o programa, detectando eventuais diferenças entre o comportamento do programa e aquele descrito pelo observador [16].

3.2 Trabalhos relacionados com o uso de verificação em CLPs

Existem vários trabalhos importantes sobre verificação de programas de CLPs, alguns deles são tratados neste capítulo. Um dos primeiros e mais importantes trabalhos na área de verificação de CLPs foi feito por Moon [34], que apresenta um método de verificação em LD. São definidas algumas regras para escrever as especificações em lógica temporal CTL (*Computation Tree Logic*) [25] para os acionamentos típicos de um CLP. Este trabalho mostra uma forma de modelagem de um programa escrito em LD no formato de um código da ferramenta de verificação de modelos SMV (*Symbolic Model Verifier*) [30], e o programa é traduzido linha a linha para esta linguagem. Utiliza uma instrução chamada *next* para simular a execução sequencial do LD na ferramenta SMV para fazer a verificação. Tem a restrição de não modelar saltos na execução do programa, ou a captura de bordas de subida/descida e temporizadores.

Outro trabalho relacionado importante, foi feito por Canet et al. [13] que utiliza a linguagem IL como entrada. É mais detalhado do que o método de Moon [34], pois cada passo do modelo, na ferramenta SMV, é um passo de execução do IL. A técnica leva a modelos que consomem mais espaço de estados na verificação do que a técnica de Moon. Assim como o trabalho de Moon, possui algumas restrições como, trabalhar apenas com variáveis booleanas, inteiros limitados e não modelar temporizadores.

O trabalho de Frey e Litz [20] faz uma abordagem geral de métodos formais na programação de CLPs. É feito um modelo genérico detalhado de um processo de projeto de controle. Neste trabalho, são usadas, para o levantamento, diferentes abordagens formais no contexto de programação de CLPs. O trabalho foca nos métodos formais para verificação e validação. Os diversos assuntos apresentados são categorizados usando três critérios: a abordagem geral para a tarefa (baseado em modelo, baseado em restrições ou sem um modelo), o formalismo (rede de Petri, autômato, sistema condição/evento) usado na descrição formal do

estado, e o método (*model-checking*, analise de alcançabilidade, prova por teorema) usado para analisar as propriedades.

Um trabalho mais completo foi realizado por Rossi e Schnoebelen [41], o qual utiliza como linguagem de entrada o LD e a ferramenta de verificação Cadence SMV, que também é outra variação do SMV. Possui algumas pequenas restrições como: todas as variáveis são booleanas (ou seja, não possui variáveis inteiras), existe apenas um programa LD rodando no CLP e cada linha de programa do LD é composta de uma parte de teste (lógica inicial de controle), seguida de uma parte de atribuição desta lógica. Possui também como característica, a modelagem de saltos na execução do programa e captura de bordas de subida/descida, assim como também pode modelar temporizadores.

O trabalho de Zoubeck [49] desenvolveu uma ferramenta que converte programas de CLP escritos em LD para uma rede de autômatos temporizados tratável pela ferramenta de verificação UPPAAL [4]. Devido a modelos grandes e complexos foram propostos algoritmos de abstração para reduzir o espaço de estados antes de se chegar ao modelo convertido em autômatos do UPPAAL. Estes algoritmos dependem das propriedades a serem verificadas. A modelagem proposta por Zoubek adota modelos separados para o programa principal e o comportamento do CLP.

O trabalho de Gourcuff et al. [23] aborda a escalabilidade do *model-checking* usando o verificador de modelos NuSMV, versão mais nova da ferramenta SMV. Para evitar, ou pelo menos limitar, a explosão combinatória, uma representação eficiente de programas de CLP foi proposta. Essa representação inclui apenas os estados que são significativos para a prova de propriedades. Este trabalho também desenvolveu um método para traduzir os programas de CLP desenvolvidos na linguagem ST para modelos no NuSMV: a representação é descrita e apresentada através de vários exemplos.

Um trabalho mais recente, e também importante, foi feito por Mokadem [33] [32] que aborda a conversão de programas de CLP escritos em Grafcet / SFC para autômatos temporizados. Este trabalho tem

como uma característica importante a melhoria do modelo de temporizador de Mader e Wupper [29], que fez a modelagem de um temporizador do tipo TON da linguagem IL e comprovou a sua correção através de prova por teorema. O modelo do temporizador TON, proposto por Mokadem, incorpora as seguintes vantagens em relação ao proposto por Mader e Wupper: só possui um canal de sincronismo contra os três propostos anteriormente, não possui os *selfloops*, tornando o modelo mais limpo, e as atualizações das variáveis são feitas internamente, isolando seu comportamento do modelo do programa principal. O trabalho de Mokadem também contempla a programação multitarefa e propõe uma solução para a abstração de estados transitórios. Por fim, faz uso de um autômato auxiliar o qual é denominado de autômato observador (para fazer a verificação por alcançabilidade, testando se o autômato chegou ao estado esperado) e aborda propriedades específicas relacionadas ao modelo (o exemplo tratado por Mokadem é de uma esteira rolante em um sistema de produção) para a verificação através de lógica temporal TCTL.

Por fim, outro trabalho interessante foi desenvolvido por Silva [44] no IME (Instituto Militar de Engenharia), que abordou a verificação de modelos aplicados a programas de CLPs, e foi uma continuação do trabalho iniciado por Oliveira [37], também do IME. Este trabalho desenvolveu um projeto de melhoria do processo de engenharia de programas de CLPs de sistemas instrumentados de segurança. Os programas considerados foram escritos em FBD e o documento básico para extração das especificações formais foi a matriz de causa e efeito. O trabalho descreve sua sequência principal, justificando a escolha do arcabouço teórico com: autômatos temporizados, matrizes de limites de diferenças e o verificador UPPAAL [4].

Como descrito anteriormente, através dos trabalhos relacionados, a maioria apresenta muitas limitações e alguns tratam de linguagens apenas textuais, como IL e ST. Os primeiros trabalhos também não levam em consideração a questão do tempo, utilizada por temporizadores. Os trabalhos mais recentes são mais completos, porém o trabalho de-

senvolvido nesta dissertação utiliza ferramentas e técnicas de verificação que podem surgir como alternativa para os trabalhos já existentes. Um dos objetivos deste trabalho é propor a verificação para sistemas de CLP, através de duas abordagens distintas, utilizando a linguagem de CLP mais usual, como o LD de entrada, e duas ferramentas de verificação. Com este trabalho, pretende-se verificar CLPs programados em LD, abrangendo todos os elementos mínimos (elementos básicos, blocos temporizadores, entre outros) descritos pela norma IEC. A construção destas cadeias de verificação, para estas duas abordagens, contam com um ponto em comum, que é a linguagem intermediária para verificação chamada FIACRE. Estas cadeias de verificação serão construídas baseadas na técnica de engenharia dirigida a modelos (MDE), através da transformação de modelos.

3.3 Engenharia dirigida a modelos (MDE)

Nesta seção é apresentada a engenharia dirigida a modelos, necessária para a transformação das linguagens inseridas nas cadeias de verificação formal deste trabalho. Para isto, é necessária a utilização dos conceitos descritos de acordo com o MDE. O termo MDE (*Model Driven Engineering*) é normalmente usado para descrever formas de desenvolvimento de sistemas em que modelos abstratos são criados e sistematicamente transformados em implementações concretas. O MDE é uma abordagem recente na comunidade de desenvolvimento de software em que os modelos são considerados como as principais entidades de todo o ciclo de vida do *software* [43]. Uma das principais iniciativas em MDE é a MDA (*Model Driven Architecture*), voltada para a portabilidade, interoperabilidade e reusabilidade, obtidas por intermédio da modelagem da operação de um sistema de maneira independente da plataforma a ser usada posteriormente para sua implementação [38].

De acordo com a MDE, a transformação de modelos pode ocorrer entre níveis iguais e diferentes de abstração, como também pode ocorrer entre mesmos domínios e domínios diferentes. A transformação é

a geração automática de um modelo destino a partir de um modelo origem. Essa transformação é definida por um conjunto de regras, que juntas, descrevem como um modelo na linguagem origem pode ser transformado em um ou mais modelos na linguagem destino. As transformações de modelos podem ser classificadas de transformação *Modelo-Modelo* e transformação *Modelo-Texto*. Na categoria *Modelo-Modelo*, a transformação pode ser diferenciada entre as seguintes abordagens: manipulação direta, orientada a estrutura, operacional, baseada em *template*, relacional, baseada em grafos e híbrida. As transformações *Modelo-Texto* são aplicadas para gerar códigos em uma determinada linguagem a partir de um modelo dependente de plataforma.

Para que aconteça a transformação entre os modelos, é preciso definir as regras de transformação do modelo que representa o transformador. Essas regras são baseadas nas estruturas dos elementos dos metamodelos de origem e de destino. O modelo de transformação recebe como entrada o modelo origem e o transforma no modelo destino. De acordo com os domínios dos metamodelos de origem e destino, o transformador de modelos pode gerar novos modelos no mesmo domínio (a fim de se obter um grau maior de especificidade em relação ao modelo original) e novos modelos em domínios diferentes (a fim de se obter reuso de modelos para criação de novos modelos em diferentes domínios). Uma aplicação pode ser gerada automaticamente de uma especificação escrita em uma linguagem textual ou gráfica de um determinado domínio de problemas [31]. A Fig. 3.1 ilustra, de forma geral, a estrutura base para a transformação de modelos. O modelo de entrada *'Ma'* deve estar conforme o seu metamodelo de entrada *'MMa'*. O mesmo ocorre com o modelo de saída *'Mb'*, conforme ao metamodelo *'MMb'*, e o modelo de transformação *'Mt'*, conforme o metamodelo *'MMt'*. Todos estes metamodelos, por sua vez, devem estar conformes a um meta-metamodelo *'MMM'*. Para garantir que uma transformação de um modelo em outro seja correta, é necessário que os metamodelos de entrada, de saída e de transformação, tenham o mesmo meta-metamodelo. Desse modo, será gerado, a partir do modelo de entrada *Ma*, um modelo de saída

Mb, de acordo com as regras de transformação escritas no modelo de transformação *Mt*.

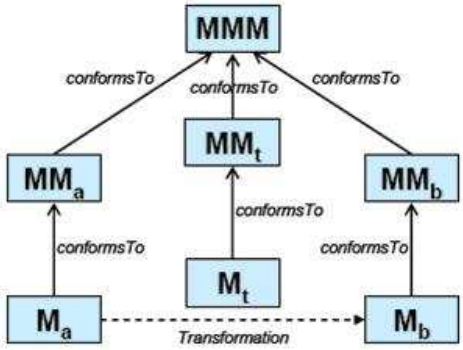


Figura 3.1: Visão geral da estrutura base para a transformação de modelos [31].

Para a transformação de modelos, é necessário utilizar uma linguagem para expressar as regras de transformação, uma linguagem bastante utilizada é a ATL. A ATL (*Atlas Transformation Language*) é uma linguagem híbrida, que possui construções imperativas e declarativas, feita para expressar transformações de modelos como requeridas pela abordagem de MDA. É descrita por uma sintaxe abstrata, um metamodelo MOF (*Meta-Object Facility*), definido pela OMG (*Object Management Group*) [38], uma sintaxe textual concreta e uma notação gráfica que permite que visões parciais de regras de transformação possam ser definidas. Um metamodelo usa MOF para definição formal da sintaxe abstrata de um conjunto de construtores de modelos. O MOF estabelece padrões para modelagem e construtores de troca de dados utilizados pelo MDA. O programa de transformações em ATL é composto de regras que definem como elementos em um primeiro modelo são mapeados para os elementos em um segundo modelo [2].

De acordo com o que foi apresentado anteriormente, devem ser especificados os elementos que pertencem a transformação dos programas na linguagem LD para FIACRE, e finalmente, para os modelos formais

(TTS e LOTOS) utilizados para a verificação. Como passo inicial para o primeiro processo de transformação, é preciso definir os metamodelos que serão utilizados para a transformação. Neste caso, o metamodelo de entrada do LD e o metamodelo de saída do FIACRE. Estes metamodelos necessitam estar em conformidade com a linguagem de transformação ATL, que utiliza o meta-metamodelo *Ecore* [12]. O *Ecore* faz parte do projeto EMF (*Eclipse Modeling Framework*), que consiste em um ambiente de modelagem para ferramentas de construção e outras aplicações baseadas em um modelo de dados estruturado.

Os metamodelos podem ser criados em um editor gráfico, como o do ambiente Topcased, do qual esse projeto faz parte, facilitando o seu desenvolvimento, ou diretamente através do ambiente Eclipse [12]. O editor *Ecore*, do ambiente Topcased, possibilita a criação de um metamodelo automaticamente, a partir de um diagrama de classes UML construído de forma gráfica, através de uma estrutura em forma de árvore. O metamodelo para o LD é gerado a partir do esquema XML da norma, integrante do projeto PLCOpen. Um trabalho de iniciação científica em curso na UFSC visa adaptar o metamodelo para do LD, de acordo com o editor *PLCOpen Editor*. O metamodelo FIACRE, utilizado no ambiente Topcased, está em constante desenvolvimento pelos membros do projeto [18]. Da mesma forma que os metamodelos dos modelos TTS e LOTOS também já foram definidos pelo projeto Topcased.

Para o processo de transformação, são necessários então: o metametamodelo (para este projeto foi utilizado o Ecore), o metamodelo de entrada LD (descrevendo apenas a sintaxe do LD, mas não o seu aspecto dinâmico), o metamodelo de saída FIACRE, os modelos de entrada (programas escritos em LD e editados na ferramenta PLCOpen Editor, salvos no formato definido no esquema XML da norma) e o modelo de transformação (através de uma linguagem de transformação, com as regras que mapeiam os elementos do metamodelo LD nos elementos do metamodelo FIACRE, que possibilitam a descrição dos aspectos dinâmicos da linguagem LD). O modelo de transformação baseado nas

regras de tradução de LD para FIACRE é tratado no Capítulo 4.

3.4 Cadeia de verificação formal

De acordo com os objetivos deste trabalho, em transformar os programas na linguagem LD para formalismos possíveis de serem utilizados nas ferramentas de verificação, é necessário definir as cadeias de verificação formal onde estas transformações irão ocorrer. Este trabalho está inserido no contexto do projeto Topcased (*Toolkit in OPen-source for Critical Application and SystEms Development*), ambiente de desenvolvimento com um conjunto de ferramentas de engenharia de sistemas para o desenvolvimento de sistemas críticos de tempo real [15]. No ambiente Topcased foram propostas cadeias de verificação, que a partir de linguagens de modelagem próprias de usuários, permitem, por transformações sucessivas, utilizar as ferramentas de verificação formal. No contexto deste trabalho, denomina-se por cadeia de verificação, a seqüência de transformações e ferramentas necessárias para realizar o processo de verificação formal de um sistema descrito em uma linguagem de usuário. A Fig. 3.2 descreve as etapas de uma cadeia de verificação genérica, típica do ambiente Topcased.

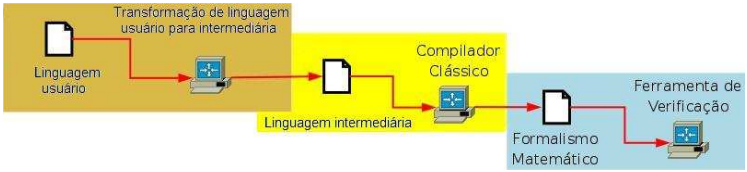


Figura 3.2: Cadeia de verificação genérica.

No exemplo da Fig. 3.2, a cadeia de verificação possui duas transformações de modelo e uma ferramenta de verificação. A primeira transformação consiste na tradução do sistema descrito em alguma linguagem de modelagem próxima do usuário (podendo ser, em alguns casos, uma descrição informal ou semi-formal) para uma linguagem intermediária. A segunda é a tradução desta linguagem para o formalismo matemático

selecionado, de acordo com a ferramenta de verificação escolhida. No caso deste trabalho, é utilizada a linguagem LD como entrada e a linguagem intermediária FIACRE. O formalismo matemático utilizado pelo ambiente Topcased pode variar conforme a ferramenta de verificação a ser utilizada. No caso deste trabalho, existem duas possibilidades, podendo ser TTS (sistema de transição temporizado) para a ferramenta TINA (*Time Petri Net Analyzer*) [46] ou LOTOS (*Language of Temporal Ordering Specification*) para a ferramenta CADP (*Construction and Analysis of Distributed Processes*) [22]. Tendo em vista que a compilação entre FIACRE e TTS/LOTOS já foram realizadas, a primeira transformação da Fig. 3.2 é justamente um dos objetivos deste trabalho, ou seja, a inserção na cadeia de verificação de uma nova transformação, tendo como linguagem de entrada o LD para a programação de CLPs. Este trabalho apresenta duas cadeias de verificação de CLPs. Os métodos de verificação a serem utilizados são baseados nas abordagens lógica (*model-checking*) e comportamental (equivalências).

Conforme apresentado anteriormente, descreve-se a seguir, as duas cadeias de verificação formal a serem utilizadas neste trabalho. A Fig. 3.3 representa a cadeia de verificação para CLPs utilizando a abordagem *model-checking*. Nesta figura, o foco deste trabalho está nos dois blocos grifados, ou seja, no tradutor LD-FIACRE e na transformação das propriedades para a lógica temporal LTL. A saída gerada pelo editor IEC de linguagens de CLP da norma (PLCOpen Editor) é um arquivo XML descrevendo os sistemas programados em CLP. Este arquivo gerado serve de entrada para o tradutor LD-FIACRE, que é um dos objetivos deste trabalho. A saída deste tradutor será um arquivo na linguagem FIACRE que servirá de base para a próxima etapa de tradução FIACRE-TTS. Esta tarefa já foi realizada e permite que a ferramenta TINA possa receber o sistema de transição temporizado (TTS) e, junto com a lógica LTL representando as propriedades, realizar a etapa de verificação através do *model-checking*.

A Fig. 3.4 representa a cadeia de verificação formal utilizando a abordagem de equivalência de modelos. De forma similar a Fig. 3.3,

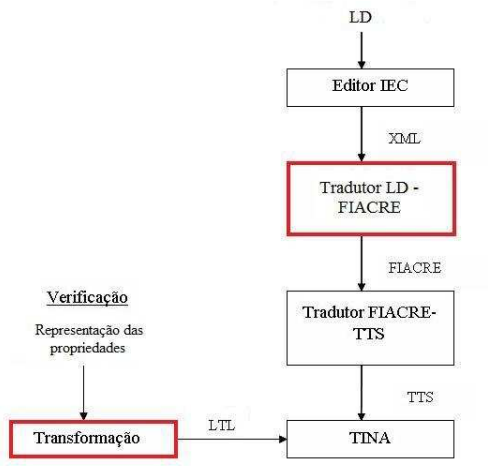


Figura 3.3: Cadeia de verificação utilizando model-checking.

nesta figura, o foco do trabalho está nos dois blocos grifados, ou seja, no tradutor LD-FIACRE e na transformação das propriedades para a linguagem FIACRE. Nesta cadeia de verificação, o objetivo é utilizar a equivalência observacional entre o modelo representando o programa em CLP e o modelo representando as especificações do programa. A saída do tradutor LD/FIACRE será um arquivo na linguagem FIACRE representando o programa, que servirá de base para a próxima etapa de tradução FIACRE-LOTOS, já existente, e que permite que a ferramenta CADP, também integrante do ambiente Topcased, possa receber o arquivo correspondente. O tradutor da representação das propriedades para FIACRE tem como saída um modelo em FIACRE representando as especificações do programa, que também servirá de base para a tradução para a linguagem LOTOS, que é a entrada da ferramenta CADP, para a verificação através de equivalência, ou através do uso de observadores. Existe também a possibilidade de gerar os autômatos (ou sistema de transição) e entrar diretamente na ferramenta CADP.

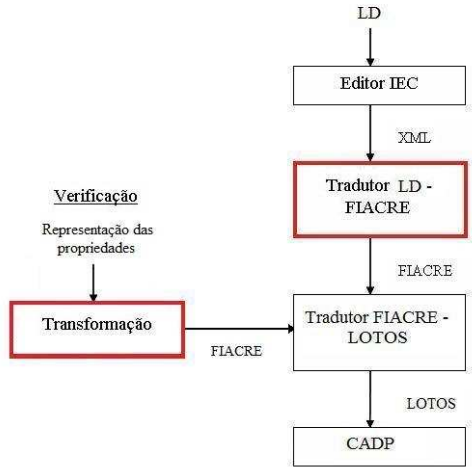


Figura 3.4: Cadeia de verificação utilizando equivalência.

3.5 Linguagem intermediária de verificação FIACRE

Nesta seção é apresentada uma linguagem intermediária dentro da cadeia de verificação de CLPs. Um dos principais objetivos deste trabalho é traduzir programas feitos na linguagem LD para a linguagem intermediária de verificação FIACRE, integrante do projeto Topcased, para então facilitar a aplicação de métodos de verificação formal. O resultado do programa em FIACRE é uma etapa intermediária e serve de entrada para os outros tradutores (FIACRE/TTS e FIACRE/LOTOS) para então ser utilizada nas ferramentas de verificação TINA e CADP, conforme visto anteriormente.

A linguagem FIACRE (*Format Intermédiaire pour les Architectures de Composants Repartis Embarques* - Formato Intermediário para Arquiteturas de Componentes Distribuídos Embarcados) foi concebida para ser uma linguagem intermediária formal entre as linguagens de modelagem de alto nível e as ferramentas de verificação que trabalham com linguagens baseadas em formalismos matemáticos (autômatos, redes de

Petri, sistemas de transição temporizados). FIACRE é um modelo intermediário formal para representar tanto os aspectos comportamentais como temporais do sistema, em particular os sistemas distribuídos e embarcados, para propósitos de verificação e simulação [9].

FIACRE é uma linguagem orientada a processos e possui as seguintes características:

- os processos são construídos por um conjunto de estados, uma lista de transições entre estes estados formadas por construções clássicas (atribuições de variáveis, construções *if-then-else, while*, composições seqüenciais), não determinísticas e para comunicação através de portas;
- os componentes descrevem a composição de processos. Um componente é construído como uma composição paralela de outros componentes ou processos que podem se comunicar por intermédio de portas ou variáveis compartilhadas. As prioridades e as restrições temporais são associadas à comunicação.

FIACRE foi concebido no âmbito de projetos relacionados com a engenharia dirigida a modelos (MDE) e é resultado da reunião de diversos parceiros, tanto das indústrias como acadêmicos. Portanto o FIACRE é projetado tanto como uma linguagem destino da ferramenta de transformação de modelo, a partir de vários modelos como AADL ou UML, como uma linguagem origem de compiladores com o objetivo final de verificação [8].

A sintaxe da linguagem FIACRE é descrita da seguinte forma:

- *Tipos*: os tipos de dados aceitos são divididos em dois grupos: tipos de base e tipos construídos. O FIACRE aceita também a definição de *alias*, ou seja, permite renomear os tipos existentes. Os tipos de base nativos são compostos por inteiro (*int*), natural (*nat*) e booleano (*bool*). O FIACRE permite ao usuário a criação de tipos específicos utilizando como base os tipos nativos: intervalos de inteiros, enumerações, estruturas, matrizes de tamanho fixo, pilhas de tamanho limitado e uniões.

- *Portas e canais de comunicação*: as portas (*port*) fazem parte da interface de um processo FIACRE. Elas são responsáveis pela comunicação e podem ser utilizadas para a troca de dados. Os canais (*channel*) são usados para definir um conjunto de tipos de dados aceitos por uma porta. Um perfil do tipo *none* sinaliza que a comunicação em questão é uma sincronização pura, sem nenhuma troca de valor.

- *Processos*: um processo FIACRE pode ser visto como uma quíntupla *Pc = (Pt, Pm, S, V, T)*, onde:

  - *Pt* é um conjunto finito de portas. Estas portas são utilizadas para efetuar a sincronização de comportamento com outros elementos do sistema (processos ou componentes);
  - *Pm* é um conjunto finito de parâmetros (formais e variáveis compartilhadas);
  - *S* é um conjunto finito de estados para o controle interno dos processos;
  - *V* é um conjunto finito de variáveis locais;
  - *T* é um conjunto de transições atômicas.

- *Comportamento*: o comportamento do processo FIACRE é descrito de forma estruturada a partir de uma lista de transições (*conjunto T*). Estas transições possuem um estado de partida e outro de chegada, e seus corpos são formados por uma estrutura de controle.

- *Comunicação*: o FIACRE permite a comunicação síncrona entre processos e/ou componentes através das portas de comunicação. Estas portas permitem a sincronização pura ou a passagem de um, ou diversos valores. Uma comunicação pode ser restringida pela opção *where*, a qual impede a sincronização se a expressão não for verdadeira. Os operadores *'?'* e *'!'* determinam o sentido da comunicação, ou seja, se a transição envia ou recebe um dado. No

caso de uma comunicação através de uma porta sobrecarregada (mais de um perfil), um campo opcional (*profile*) é utilizado para definir o perfil desejado.

- *Componente*: um componente FIACRE compreende a composição paralela de processos ou de outros componentes. São construídos a partir da instanciação de outros elementos do sistema (processos ou componentes), especificando as suas interconexões (canais de comunicação) e fornecendo uma relação de prioridades entre elas. Um componente pode ser visto como uma quíntupla *Cmp = (Pt, Pm, V, C, Pr, Cm)*, onde:
  - *Pt* é um conjunto finito de portas;
  - *Pm* é um conjunto finito de parâmetros (formais e variáveis compartilhadas);
  - *V* é um conjunto finito de variáveis locais;
  - *C* é um conjunto finito de portas locais (*port*) associadas a restrições temporais (*is*). Estas portas locais também são utilizadas para interligar as instâncias que compõem o componente e são chamadas de canais de comunicação porque permitem a comunicação interna no componente;
  - *Pr* é um conjunto finito de prioridades;
  - *Cm* é uma composição paralela de instâncias de processos. A composição descreve a interação entre as instâncias que compõem o componente.

- *Composição*: a comunicação síncrona presente em FIACRE é resultado da composição paralela de um conjunto de instâncias. A composição promove um determinado tipo de comunicação. Esta comunicação pode ser descrita por intermédio de um operador de composição paralela:
  - *par* (composição paralela): este operador realiza a execução simultânea das ações observáveis entre processos e/ou com-

ponentes. As ações são ditas observáveis quando são relacionadas às portas declaradas pelo operador *par*.

- *sync* (sincronização): este operador é uma simplificação do operador de composição paralela, quando todas as ações são observáveis e representa a sincronização total entre os processos ou componentes implicados. Neste caso, o operador *sync* é uma simplificação do operador *par* quando todas as portas que fazem parte da interface dos processos ou componentes são locais.
- *shuffle* (paralelismo puro) : este operador também é uma simplificação do operador *par* quando não existe nenhuma ação observável, ou seja, representa uma operação de composição paralela *par* com um conjunto vazio de portas [42].

Um programa FIACRE, para ser completo, necessita de um conjunto de tipos, canais de comunicação, processos e componentes. A Fig. 3.5 ilustra um exemplo de código em FIACRE que implementa a exclusão mútua entre três usuários. O processo usuário (*user*) começa do estado *idle* e o recurso do estado *free*. Ambos, após a transição *enter*, passam para o estado *busy*, voltando a ficar livre após a transição *leave*. A diferença entre os processos *user* e *mutex* é que o usuário requisita o recurso, e o recurso é alocado ao usuário. No componente *main* é feita a comunicação entre os processos, através dos *ports enter* e *leave* e da composição *par* dos processos *user* e *mutex*. No *port* é também definido o tempo de cada transição, que neste caso é *[0,...[*.

3.6 Formalismos e ferramentas para a verificação

De acordo com o que foi apresentado neste capítulo, a última transformação da cadeia de verificação gera como saída os formalismos matemáticos para as ferramentas de verificação. As duas cadeias de verificação propostas apresentam dois formalismos diferentes para

```
process user[enter : out int, leave : none](self : int) is
      states idle, busy
      init to idle

      from idle enter!self; to busy
      from busy leave; to idle


process mutex[enter : in int, leave : none] is
      states free, busy
      var current : int := 0
      init to free

      from free enter?current; to busy
      from busy leave; to free


component main is
      port
            enter : int in [0, ...[ ,
            leave :  none  in [0, ...[
      par
            enter,leave -> mutex[enter,leave]
       ||         enter,leave -> par
              ->         user[enter,leave](0)
             || ->       user[enter,leave](1)
             || ->       user[enter,leave](2)
                end
      end
main
```

Figura 3.5: Exemplo de código em FIACRE [18].

as ferramentas de verificação TINA e CADP, integrantes do projeto Topcased. Para a cadeia de verificação utilizando o *model-checking*, o formalismo utilizado é o TTS que serve de entrada para o TINA. Para a cadeia de verificação utilizando equivalências o formalismo é o LOTOS para a ferramenta CADP.

3.6.1 Formalismo TTS e a ferramenta TINA

O compilador FIACRE/TTS, pertencente a cadeia de verificação, desenvolvido inicialmente por Saad [42] e constantemente aprimorado pelo grupo de pesquisa do LAAS da França, utiliza o formalismo matemático TTS como alvo para ser utilizado na ferramenta TINA. Este formalismo foi selecionado porque é muito utilizado para verificar formalmente sistemas que são caracterizados como: concorrentes, assíncronos, distribuídos, paralelos ou não determinísticos. O TTS pode ser considerado como uma generalização do sistema de transições de base através da associação de tempos mínimos e máximos para as transições.

Um TTS é composto basicamente por seis parâmetros: um conjunto finito de variáveis, um conjunto de estados, um sub-conjunto de estados iniciais, um conjunto de transições (as restrições temporais asseguram que as transições serão disparadas somente no intervalo de tempo permitido), um atraso inicial mínimo para cada transição e um atraso máximo para cada transição.

É possível interpretar o TTS como uma extensão da Rede de Petri Temporal (TPN) estruturada em duas partes: controle e dados. A parte de controle é descrita pela TPN comum e descreve os encadeamentos de eventos e atividades. Assim, as variáveis que são importantes para o controle do sistema são representadas como lugares da TPN empregada. A parte de dados descreve as variáveis que não pertencem à estrutura de controle do sistema. Os cálculos, que são realizados sobre a estrutura de dados, são representados associando expressões condicionais e ações às transições.

O compilador FIACRE/TTS, da cadeia de verificação, gera como saída um diretório (*.tts*) com dois arquivos (*.net e .c*). Após a tradução, se o modelo utiliza estruturas de dados externas à TPN (*unions, queues, booleans*, etc.), é necessário compilar o arquivo *.c* para ser utilizado pela ferramenta TINA. Depois de compilado, é possível executar a verificação formal, juntamente com a ferramenta de verificação SELT, dentro do ambiente TINA [7]. A Fig. 3.6 ilustra como ocorre a verificação no ambiente TINA a partir de um modelo FIACRE. Com a descrição em FIACRE, a partir da transformação LD-FIACRE, é possível traduzir para o formalismo TTS por intermédio do compilador desenvolvido inicialmente por Saad [42]. Após a tradução do modelo para TTS, o TINA é utilizado para construir uma abstração adequada do grafo de alcançabilidade do sistema como um sistema de transição *Kripke* (utilizado para a descrição de sistemas concorrentes ou reativos, que representam sistemas com comportamento infinito), com a extensão *.ktz*. Este grafo é confrontado com as propriedades formuladas em LTL (arquivo *'.ltl'*) pelo *model-checker* SELT, que pertence ao ambiente TINA [7].

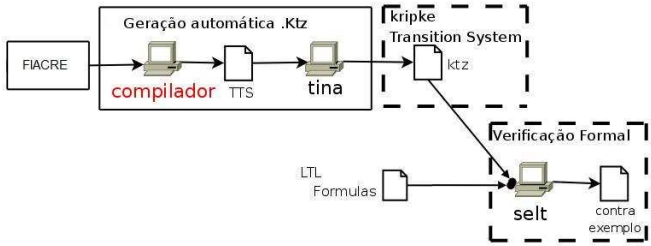


Figura 3.6: Verificação utilizando o ambiente TINA [42].

O TINA é uma ferramenta para edição e análise de redes de Petri e TPN, desenvolvido no grupo de pesquisa do LAAS/CNRS da França. O TINA tem uma grande importância pelo fato de utilizar e ter suporte para ferramentas de verificação, além de ser capaz de editar e analisar outras extensões, como o TTS. As diferentes ferramentas que constituem o ambiente podem ser utilizadas isoladamente ou em conjunto. A ferramenta de verificação que integra o TINA é o SELT, para verificar propriedades específicas, além de acessibilidade, limitação, ausência de *deadlock* e vivacidade, já verificadas pelo TINA. A ferramenta SELT é um verificador de modelo para uma versão enriquecida de Estado/Evento LTL. No caso de não satisfazer a propriedade, o SELT é capaz de gerar uma sequência legível de contra-exemplo em uma ou mais formas utilizáveis pelo simulador do TINA, a fim de executá-lo passo a passo [7].

Para expressar as propriedades a serem verificadas através de fórmulas de lógica, utilizadas no SELT, é utilizado um subconjunto da lógica temporal LTL. Estas fórmulas são construídas a partir de um conjunto de variáveis proposicionais e operadores lógicos usuais como os seguintes, de acordo com a sintaxe utilizada no SELT: implicação ($\Rightarrow$), negação (-), conjunção ($\wedge$), disjunção ($\vee$) e constantes (*true e false*).

Além desses, o subconjunto também possui operadores temporais:

- estado futuro ($\Diamond$);

- todos os estados futuros ([]).

Para utilização da lógica é necessário um refinamento das propriedades nas fórmulas, substituindo as variáveis proposicionais pelos nomes dos estados definidos nos modelos das propriedades a serem verificadas [42].

3.6.2 Formalismo LOTOS e a ferramenta CADP

Para a cadeia de verificação, utilizando a equivalência de modelos, o formalismo adotado é o LOTOS, que serve de entrada para a ferramenta CADP, através do compilador FIACRE/LOTOS. O LOTOS é uma técnica de descrição formal padronizada pela ISO (*International Organization for Standardization*) para especificação de protocolos de comunicação e sistemas distribuídos. O LOTOS é composto por duas sub-linguagens. A primeira parte é o LOTOS básico, que representa o modelo estritamente comportamental do sistema. Nesse modelo, o sistema é descrito como um conjunto de processos, os quais são sincronizados mediante a execução de ações previamente especificadas em pontos de interação compartilhados e sua representação é baseada na álgebra de processos. A parte do LOTOS completo inclui a definição dos tipos de dados envolvidos nas interações entre os processos, com base na teoria de tipos de dados algébricos abstratos. O LOTOS possui como limitação a não representação explícita do tempo [10].

O ambiente utilizado para a linguagem LOTOS é o CADP, que é um conjunto de ferramentas empregado em engenharia de protocolos, com o objetivo de promover a compilação, simulação, verificação formal e teste de descrição de protocolos escritos, principalmente, na linguagem LOTOS. O CADP oferece um amplo conjunto de funcionalidades como compilação de especificações descritas em LOTOS, verificação de sistemas comunicantes, validação e testes de protocolos, além de outras que vão desde a simulação passo-a-passo até o *model-checking*. Além disso, o CADP possui como característica importante a variedade de ferramentas de verificação de equivalência (minimização e comparações

de relações de bisimulação).

O CADP inclui ainda varias funcionalidades para o projeto de sistemas assíncronos como: geração de código, prototipagem rápida, verificação de estado explícito, geração de casos de teste e avaliação de desempenho. O CADP aceita como entrada redes de comunicação de autômatos ou linguagens de especificação de alto nível, como o LOTOS. Muitas ferramentas do CADP operam também com um LTS (*Labeled Transition Systems*), que são representados de forma explícita, como arquivos binários compactos codificados no formato BCG (*Binary Coded Graphs*), ou implicitamente, como implementações de programas em C através de uma aplicação de interface. Basicamente, três formas de verificação são suportadas pelo CADP: *visual checking* (inspeção gráfica de um LTS), *model checking* e verificação por equivalência (comparação de dois arquivos LOTOS ou LTS). A ferramenta de verificação CADP não trata restrições temporais explícitas [22].

3.7 Conclusão do capítulo

Foram apresentados neste capítulo os conceitos relacionados a modelagem e verificação formal de CLPs. Inicialmente, foi discutida a verificação aplicada a CLPs, juntamente com as abordagens de verificação que serão aplicadas neste trabalho e os trabalhos relacionados com a área, contextualizando o assunto. Também foi apresentada a transformação de modelos, necessária para a tradução das linguagens LD para FIACRE, inseridas na cadeia de verificação. Foram descritas as cadeias de verificação formal deste trabalho, com as duas abordagens: *model-checking* e equivalências. Os elementos que constituem essa cadeia como a linguagem intermediária FIACRE e os formalismos para as ferramentas de verificação também foram apresentados.

A partir dos conceitos relacionados aos CLP e suas formas de verificação e modelagem, juntamente com as definições de transformação de modelos, é possível caracterizar a transformação dos programas descritos na linguagem de entrada LD, para CLPs, para a linguagem inter-

mediária de verificação FIACRE. O próximo capítulo irá tratar basicamente das modelagens que compõem os programas em LD e as regras de tradução de LD para FIACRE.

Capítulo 4

Transformação de programas em LD para FIACRE

O Capítulo 3 mostrou, que neste trabalho, a verificação de programas é realizada por duas cadeias de verificação formal, baseada em transformação de modelos. Neste capítulo, é apresentada a etapa de transformação de modelos de programas de CLP escritos em LD para a linguagem intermediária FIACRE, a ser utilizada na cadeia de verificação. Primeiramente, é descrita a modelagem do comportamento das principais construções do LD, juntamente com as regras de tradução da linguagem LD para a linguagem FIACRE e os códigos em FIACRE. Também é apresentado neste capítulo um exemplo de programa de CLP escrito em LD com elementos básicos e bloco funcional, e a sua tradução para a linguagem FIACRE.

4.1 Modelagem de um programa de CLP e regras de tradução de LD para FIACRE

Nesta seção é apresentada a modelagem dos diversos elementos e construções encontrados em programas na linguagem LD. Estes modelos dos elementos de programação em LD servem como base para a definição das regras de transformação para FIACRE, que também serão apresentadas nas subseções seguintes.

4.1.1 Modelo base do CLP

A execução de um programa de CLP segue um ciclo que apresenta uma leitura de entradas oriundas do ambiente externo, execução dos cálculos descritos no programa e escrita das saídas em direção ao ambiente externo. No modelo adotado neste trabalho, a parte do programa que contém temporizações (blocos funcionais com tempo) será modelada como um bloco separado, de forma concorrente com o modelo do ciclo de execução. O restante do programa (elementos básicos sem tempo) faz parte do modelo do ciclo de execução. O CLP se comunica através de uma interface de entrada/saída, com um ambiente externo, que corresponde geralmente a planta industrial que deve ser controlada. Esta interface de entrada e saída é feita através de processos chamados de *glue*, como por exemplo, os que representam os sensores e os atuadores. O conceito de *glue* foi inicialmente criado para gerenciar a comunicação assíncrona entre as *threads*, ou os processos, a partir da utilização da comunicação síncrona entre cada processo e o processo *glue*. O processo *glue* envia (ou recebe) uma mensagem para uma *thread* no momento destinado ao envio (ou recebimento, respectivamente) e utiliza como parâmetros o valor dos dados das portas de entrada e saída [9].

A Fig. 4.1 ilustra a visão geral adotada para o modelo de um programa em LD, de um CLP em relação com o ambiente externo (ou

planta), utilizando um processo *glue* que corresponde aos sensores e atuadores. Nas subseções seguintes serão descritos cada um dos elementos que compõem um programa de CLP, juntamente com as regras de tradução para FIACRE. A descrição do processo *glue* e da planta serão objetos de estudo do capítulo seguinte. A representação em FIACRE do *glue* e do programa LD controlando a planta também serão apresentados.

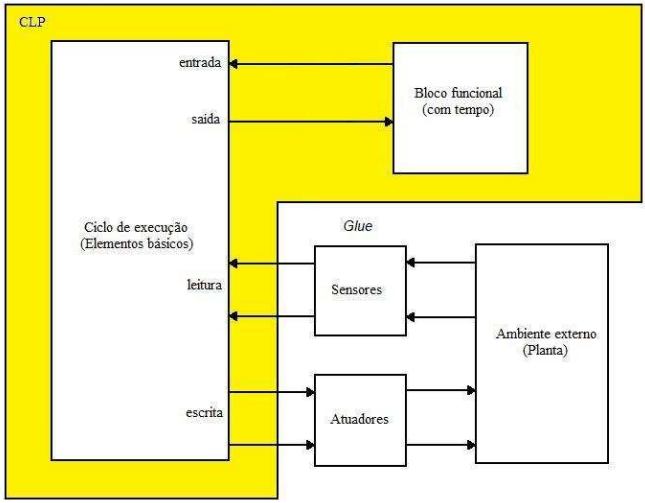


Figura 4.1: Visão geral do modelo comportamental de um programa em LD.

O programa do CLP em FIACRE será representado por um processo correspondendo ao ciclo de execução (que utiliza apenas elementos básicos sem tempo envolvido). Os blocos funcionais com temporização serão representados por outros processos, que podem ser executados em paralelo, para que não haja interrupção no processo do ciclo de execução. Para entender melhor o comportamento destes, os modelos dos elementos do LD serão descritos na forma de máquina de estados.

Em FIACRE, o sistema é representado na forma de um componente principal que é o resultado da composição dos processos correspondentes ao CLP, à planta e à *glue*. O operador *par* da linguagem

FIACRE é o elemento que permite esta composição. A composição segue a representação abaixo:

```
component main ...
.
.
.
par
  par
    par
      processo ciclo_execução
      ||
      processo bloco_funcional
    end
    ||
    processo planta
  end
  ||
  processo glue
end
```

4.1.2 Modelo do ciclo de execução

O comportamento do ciclo de execução de um CLP é formado basicamente por três etapas: leitura, execução e escrita. Primeiro, as variáveis de entrada, oriundas da planta, são lidas e seus valores são armazenados na memória. A seguir, ocorre a etapa de execução do programa, calculando os valores de saída, que também são armazenados na memória, para utilização no ciclo seguinte e/ou no controle da planta. O ciclo de execução do programa LD é assumido ser periódico. Apesar desta escolha, o ciclo poderia ser também comandado por eventos. A Fig. 4.2 caracteriza o modelo do comportamento geral, de um ciclo de execução, baseado no modelo apresentado por Mokaden et al. [33]. O sistema começa de um estado ocioso, a seguir ocorre o início de leitura dos dados de entrada. Após a leitura dos dados, ocorre o fim da leitura e início da execução do programa, no qual encontram-se as lógicas de controle. Finalizando esta execução, ocorre a atualização dos dados de saída, e depois volta ao estado inicial para esperar o reinício do ciclo,

no próximo período.

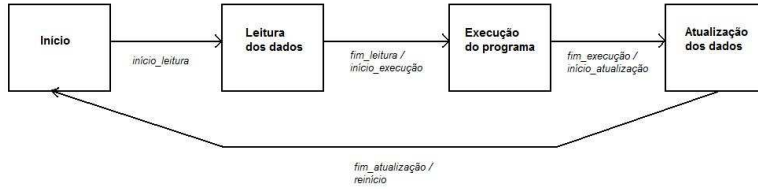


Figura 4.2: Modelo abstrato da sequência do ciclo de execução do LD.

Outra forma de representar o ciclo de execução de acordo com este trabalho, na forma de autômatos, encontra-se na Fig. 4.3. O sistema começa de um estado inicial, a seguir ocorre a transição de leitura de dados de entrada (*?dado_entrada*). Após a leitura dos dados, ocorrem as transições de execução do programa do CLP, onde encontram-se as lógicas de controle, com os elementos básicos (*rungs* com contatos e bobinas), representadas pelo pontilhado. Finalizando a execução, ocorre a transição de escrita de dados de saída (*!dado_saida*). Depois de atualizar os dados, volta-se ao estado inicial, por intermédio da transição de reinicio.

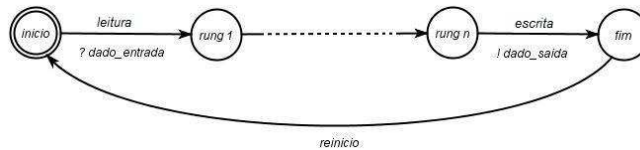


Figura 4.3: Modelo específico do comportamento do ciclo de execução do LD.

Na linguagem FIACRE, o ciclo de execução é representado por um processo periódico, da mesma forma que no modelo, ocorrendo inicialmente a leitura dos dados provenientes do modelo do ambiente externo para utilização no programa. A leitura dos dados é feita pelo recebimento de uma mensagem com o valor da variável, indicada na transição pelo símbolo *'?'*. A partir do estado *inicio*, ocorre a transição chamada

leitura (que representa a comunicação na qual o *dado_entrada* é composto pela variável de entrada que deve ser lida), após esta transição, o estado correspondente é o primeiro *rung* (*rung_1*). No fim do processo de execução do programa de CLP (estado *rung_n*), ocorre a escrita dos dados provenientes do programa para utilização no ambiente externo. A escrita dos dados é feita pelo envio de uma mensagem com o valor da variável e ocorre na transição com o símbolo *'!'*. A transição nomeada *escrita* representa a comunicação e *dado_saida* é a variável de saída que deve ser enviada. Quando o sistema possui mais de uma variável, que precisa ser lida e enviada para o ambiente, são necessárias várias transições de forma sequencial com um intervalo de tempo instantâneo, ou seja, [0,0] (podendo ser também de uma só vez, utilizando um vetor com os valores), para que todos os valores sejam sempre atualizados no mesmo instante. Após a escrita dos dados, atinge-se o estado intermediário chamado *fim*, e a partir deste, através da transição temporizada *reinicio*, volta-se para o estado inicial. Em FIACRE o ciclo de execução, é apresentado da seguinte forma (os pontos no código representam o restante das instruções, como por exemplo, declarações das transições, variáveis e estados):

```
process ciclo_execucao ...
.
.
.
from inicio
  leitura ?dado_entrada;
    to rung_1
.
.
.
from rung_n
  escrita !dado_saida;
    to fim
from fim reinicio; to inicio
```

Na etapa de execução do programa, entre a leitura e a escrita, ocorre o detalhamento dos *rungs* do LD, podendo ocorrer a subdivisão

em vários estados, de acordo com o números de *rungs*. Esta etapa é descrita nas subseções seguintes sobre contatos, bobinas e *rungs* de um programa em LD com elementos básicos. Os códigos completos em FIACRE estão nos Apêndices desta dissertação.

4.1.3 Modelos dos elementos básicos do LD

Nas próximas subseções serão apresentadas as modelagens dos diversos tipos de contatos e bobinas que constituem os elementos básicos de programas de LD.

4.1.3.1 Contatos

Os contatos simples (estáticos) podem ser classificados como NA (normalmente aberto) e NF (normalmente fechado). A Fig. 4.4 representa o contato NA, onde *'L'* representa o valor do lado esquerdo do contato, *'A'* a variável que representa o contato e *'R'* o valor do lado direito do contato.

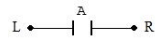


Figura 4.4: Contato NA.

Para os contatos NA, de acordo com a Fig. 4.4, o comportamento pode ser representado da seguinte forma:

```
R:= L and A.
```

O valor do lado esquerdo do contato (L) é copiado para o lado direito (R), quando a variável *'A'* for ativada. Para a modelagem, o contato NA pode ser simplificado, tornando-se apenas o valor da variável *'A'* (considerando que o lado esquerdo do contato está sempre alimentado), pois é ativado com o valor booleano *true*.

A Fig. 4.5 representa o contato NF, de forma idêntica a representação do contato NA, *'L'* representa o valor do lado esquerdo do contato NF, *'A'* a variável que representa o contato e *'R'* o valor do lado direito do contato.

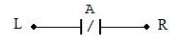


Figura 4.5: Contato NF.

Para os contatos NF, de acordo com a Fig. 4.5, o comportamento pode ser representado da seguinte forma:

```
R:= L and not A.
```

O valor do lado esquerdo do contato NF (L) é copiado para o lado direito (R), quando a variável *'A'* estiver em nível lógico zero. Para a modelagem, o contato NF pode ser simplificado, tornando-se apenas a negação da variável (considerando o lado esquerdo do contato sempre ativo), ou seja, funciona de forma contrária ao NA, pois é ativado com o valor booleano *false*.

Os contatos especiais sensíveis a bordas (subida e descida) possuem um comportamento diferente dos contatos simples. O contato sensível a borda de subida (*positive transition-sensing contact*), ou simplesmente contato P, só será ativado se no seu estado anterior (valor da variável no ciclo anterior) estiver desativado, ou seja, tem que ocorrer uma transição lógica de zero para um. Em todas as demais situações ele permanece desativado. A Fig. 4.6 representa o contato P, onde *'L'* representa o valor do lado esquerdo do contato P, *'A'* a variável que representa o contato e *'R'* o valor do lado direito do contato. As Tabelas 4.1 e 4.2 representam as lógicas de como foram criadas as expressões lógicas para ativar os contatos especiais sensíveis a bordas de subida e descida. Na tab. 4.1, R representa o valor do lado direito do contato (resultado da ativação), *A(i)* representa o valor atual da variável do contato, *A (i-1)* representa o estado anterior da variável (que deve ser armazenado) e L o valor do lado esquerdo do contato (que deve estar sempre ativo).

A Tabela 4.1 mostra como resultado para a ativação do contato P a seguinte expressão booleana:

```
R :=  L and not A(i-1) and A(i).
```

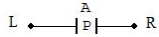


Figura 4.6: Contato P.

Tabela 4.1: Tabela lógica do contato P.

L	*A (i-1)*	*A (i)*	*R*
true	false	false	**false**
true	false	true	**true**
true	true	false	**false**
true	true	true	**false**

A variável *A* tem o seu estado inicial em *false* e deve ser sempre armazenada depois da operação descrita acima, pois é necessário guardar o valor do ciclo anterior, ou seja:

```
A(i-1):= A(i).
```

O contato sensível a borda de descida (*negative transition-sensing contact*), ou simplesmente contato N, tem um comportamento inverso ao contato P, pois só será ativado se no seu estado anterior estiver ativado, e no ciclo seguinte desativado, ou seja, tem que ocorrer uma transição lógica de verdadeiro para falso. A Fig. 4.7 representa o contato N, de forma idêntica ao contato P, *'L'* representa o valor do lado esquerdo do contato N, *'A'* a variável que representa o contato e *'R'* o valor do lado direito do contato. Na tab. 4.2, de forma similar a tabela anterior, *R* representa o valor do lado direito do contato (resultado da ativação), *A(i)* representa o valor atual da variável do contato, *A(i-1)* representa o estado anterior da variável (que foi armazenado) e *L* o valor do lado esquerdo do contato (que deve estar sempre ativo).

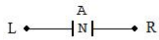


Figura 4.7: Contato N.

Tabela 4.2: Tabela lógica do contato N.

L	*A (i-1)*	*A (i)*	*R*
true	false	false	**false**
true	false	true	**false**
true	true	false	**true**
true	true	true	**false**

A Tab. 4.2 mostra como resultado para a ativação do contato N a seguinte expressão booleana:

```
R :=  L and A(i-1) and not A(i).


A(i-1):= A(i).
```

4.1.3.2 Bobinas

As bobinas simples (momentâneas, ou seja, sem retenção) da mesma forma que os contatos simples, podem ser classificadas como bobinas NA e NF. A bobina NA tem a lógica de entrada do *rung* (anterior à bobina) atribuída à sua variável. A Fig. 4.8 representa a bobina NA, onde *'X'* representa a lógica de entrada dos contatos do *rung* para ativação da bobina, *'A'* é a variável que representa a bobina e os estados representam os *rungs* do programa em LD. A Fig. 4.9 representa a modelagem do comportamento da bobina NA.

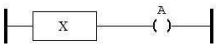


Figura 4.8: Bobina NA no rung.

Para a bobina NF, a única diferença em relação a bobina NA é a negação da lógica de entrada na atribuição para a bobina, pois a bobina NF é ativada com valor booleano *false*, onde a variável *'A'* representa

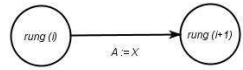


Figura 4.9: Comportamento da bobina NA.

a ativação da bobina. A Fig. 4.10 representa a bobina NF e a Fig. 4.11 representa a modelagem do comportamento da bobina NF.

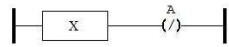


Figura 4.10: Bobina NF no rung.

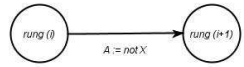


Figura 4.11: Comportamento da bobina NF.

Para as bobinas especiais *set e reset* o tratamento deve ser diferente, pois uma vez ativadas, as bobinas especiais (*set* para o caso de manter ativada e *reset* para manter desativada) permanecem acionadas nos próximos ciclos de varredura, independente do valor da lógica de entrada, ou seja, elas recebem um valor que fica retido na memória. Elas só terão o seu valor alterado quando um sinal acionar diretamente a variável. A Tab. 4.3 representa a lógica para ativar a bobina *set*. A Fig. 4.12 representa a bobina *set*, onde *'X'* representa a lógica de entrada dos contatos para ativação da bobina e *'A'* é a variável que representa a bobina set. A Fig. 4.13 representa a modelagem do comportamento da bobina *set*.

A Fig. 4.14 representa a bobina *reset*, onde *'X'* representa a lógica de entrada dos contatos para ativação da bobina e *'A'* é a variável que representa a bobina. A Tab. 4.4 representa a lógica para ativar a bobina *reset* e a Fig. 4.15 representa a modelagem do comportamento desta.

Em FIACRE, de acordo com a modelagem, para as bobinas es-

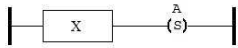


Figura 4.12: Bobina set no rung.

Tabela 4.3: Tabela lógica da bobina set.

X	***A (atual)***	***A (saída)***
false	false	**false**
false	true	**true**
true	false	**true**
true	true	**true**

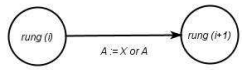


Figura 4.13: Comportamento da bobina set.

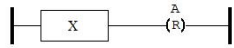


Figura 4.14: Bobina reset no rung.

Tabela 4.4: Tabela lógica da bobina reset.

X	***A (atual)***	***A (saída)***
false	false	**false**
false	true	**true**
true	false	**false**
true	true	**false**

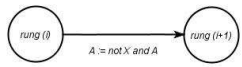


Figura 4.15: Comportamento da bobina reset.

peciais *set e reset*, a forma de escrever é feita da seguinte maneira, respectivamente:

```
%set:
from rung_i ação; A:= X or A; to rung_i+1;
%reset:
from rung_i ação; A:= not X and A; to rung_i+1.
```

A partir de um estado *'rung_i'*, ocorre uma ação instantânea (o tempo da ação é declarado no componente) com a atribuição da lógica de ativação na variável que representa a bobina, e vai para o próximo estado *'rung_i+1'*.

4.1.4 Programa em LD composto por rungs com elementos básicos

Os elementos básicos que compõem um *rung* do LD são: contatos simples estáticos, bobinas simples momentâneas, bobinas especiais com retenção (*set e reset*) e também contatos especiais sensíveis a bordas (subida e descida). Para a modelagem dos *rungs*, os contatos são representados por uma lógica de entrada e as bobinas por uma variável de saída. A Fig. 4.16 ilustra os elementos básicos em um *rung* do programa no LD, com destaque para a lógica de entrada, através dos contatos (*a, b e c*), ligados a uma bobina (*variável_saída*). A Fig. 4.17 representa a modelagem dos *rungs* de um programa LD com elementos básicos, onde um *rung* terá sempre uma lógica de controle como entrada (contato) que será armazenada em uma variável de saída (bobina). Cada estado representa um *rung* do LD, onde *i* representa o *rung* atual, *i+1* representa o próximo *rung* e a transição corresponde a passagem de um estado para outro, levando em conta a lógica entre as variáveis para atribuir um valor à variável de saída.

Em FIACRE, neste caso (rungs com apenas elementos básicos), existirá um estado por *rung*, com apenas uma transição atribuindo a lógica de controle do *rung* atual para a variável de saída. A partir do estado que representa o *rung*, ocorre uma transição que atribui a lógica de entrada para uma variável, de forma instantânea. A *'ação'* deve ser

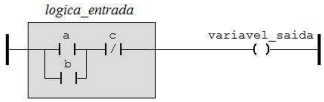


Figura 4.16: Elementos básicos em rung do LD.

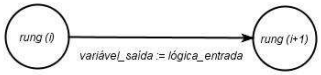


Figura 4.17: Comportamento dos elementos básicos em rungs do LD.

declarada no componente com o intervalo *[0,0]*. A lógica de controle composta pelos operadores lógicos *and e or*, definida em LD a partir do posicionamento dos elementos, é descrita textualmente em FIACRE seguindo esta lógica. De acordo com a sintaxe FIACRE, para o caso de elementos básicos, contatos e bobinas simples, um *rung* é representado por:

```
from rung_i ação; variavel_saida:= logica_entrada; to rung_i+1.
```

Os programas em LD podem conter *jumps* nos *rungs*, identificados por *labels*. A Fig. 4.18 ilustra um exemplo de programa com um *jump*, onde o contato *'A'*, quando é ativado, a execução do programa salta para o *label* chamado *'INIT'*, executa as instruções até o *label 'RETURN'* e retorna para o *rung 2*, que é o próximo a ser executado.

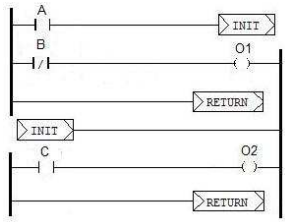


Figura 4.18: Exemplo de programa em LD com jump no rung.

Em FIACRE, quando existe um *jump* no *rung* do programa em LD, a condição de ativação do *label* é feita por um *'if'*, que se for

verdadeiro vai para o estado com o nome do *label*, senão segue para o próximo *rung*. O exemplo da Fig. 4.18 é representado em FIACRE da seguinte forma:

```
from rung1 acao1;
        if A then INIT
        else to rung2 end
from rung2 acao2;
        O1:= not B;
                to RETURN

from INIT acao3;
        O2:= C;
          to rung2
```

Os tipos de transição e os tipos das variáveis são definidos no cabeçalho do processo FIACRE. As condições iniciais das variáveis e a declaração do intervalo de tempo de cada transição são definidos no componente FIACRE, inclusive o tempo de reinício do ciclo de execução.

De forma geral, a lógica de entrada na modelagem é representada pela ligação entre os contatos no programa LD. Se os contatos estiverem ligados em série, então ocorre uma lógica booleana *'and'* entre as variáveis. Se os contatos estiverem ligados em paralelo, então ocorre uma lógica booleana *'or'* entre as variáveis. De acordo com a Fig. 4.16, a lógica de entrada dos *rungs* pode ser representada de forma resumida da seguinte forma:

- Se *lógica_entrada* é contato NA, então em FIACRE é representado por uma *variável*;
- Se *lógica_entrada* é contato NF, então em FIACRE é representado por *not variável*;
- Se *lógica_entrada* é contato P, então em FIACRE é representado por *not variável(i-1) and variável(i)*;
- Se *lógica_entrada* é contato N, então em FIACRE é representado por *variável(i-1) and not variável(i)*;

- Se *lógica_entrada* é *lógica_entrada(1)* ligada em série com *lógica_entrada(2)*, então em FIACRE é representado por *lógica_entrada(1) and lógica_entrada(2)*;
- Se *lógica_entrada* é *lógica_entrada(1)* ligada em paralelo com *lógica_entrada(2)*, então em FIACRE é representado por *lógica_entrada(1) or lógica_entrada(2)*.

De forma geral, para os elementos de saída dos *rungs*, as regras podem ser representadas da seguinte forma:

- Se existe um *rung* no LD, então em FIACRE existe um estado com a sua transição;
- Se tem bobina NA, então em FIACRE é representado por *variável := lógica_entrada*;
- Se tem bobina NF, então em FIACRE é representado por *variável := not lógica_entrada*;
- Se tem bobina set, então em FIACRE é representado por *variável := lógica_entrada or variável*;
- Se tem bobina reset, então em FIACRE é representado por *variável := not lógica_entrada and variável*;
- Se existe um contato ligado em um *label (jump)* no LD, então em FIACRE é representado pela condição *if (contato) then (vai para o estado do label, executa as instruções do 'jump' e volta para o label) else (segue para o próximo rung)*.

4.1.5 Modelo do ciclo de execução com blocos funcionais

Para o caso de programas em LD, nos quais existem blocos funcionais (do tipo temporizadores e contadores) nos *rungs*, é necessário passar por uma etapa de atualização das variáveis de entrada (IN) e de saída (Q) dos blocos funcionais. Isto é feito pelo sincronismo entre o processo do ciclo de execução e o do bloco funcional, com o envio de

mensagem (*!*) do valor de entrada do bloco (*IN*) do processo ciclo de execução para o processo do bloco funcional. Também é necessária uma etapa de sincronização para o recebimento (*?*) pelo processo ciclo de execução do valor de saída do bloco (*Q*), oriundo do processo do bloco funcional. Cada processo do bloco funcional com tempo, gera no processo ciclo de execução um envio e recebimento de mensagem com os valores de entrada e saída dos blocos funcionais.

A modelagem proposta é dividida em duas partes: a primeira delas durante o processo ciclo de execução permite a atualização das variáveis de entrada e saída do bloco funcional. A segunda parte é referente ao funcionamento de fato do bloco (por exemplo, TON, TOF e CTU), que na modelagem é representada por um processo que funciona em paralelo ao processo ciclo de execução. Um *rung* que contém um bloco funcional, no processo ciclo de execução, é modelado por transições que permitem a execução paralela dos blocos: uma representa a parte de entrada do bloco, com a lógica de ativação atribuída à variável de entrada do bloco funcional, e a outra representa a saída do bloco funcional, com o valor da variável de saída do bloco atribuída para uma variável do programa. Entre estas duas transições ocorre a atualização dos dados do bloco com o processo do bloco funcional. Esta etapa é necessária para a atualização dos dados no ciclo de execução, e serve para qualquer tipo de bloco funcional com tempo. A Fig. 4.19 ilustra um *rung* do programa no LD com um bloco funcional, com destaque para a lógica de entrada ligada no bloco, por intermédio dos contatos (*a, b e c*), e a saída do bloco ligada a uma bobina (*variavel_saida*). A Fig. 4.20 ilustra parte do modelo do ciclo de execução de um programa em LD que contém um bloco funcional. A variável *IN* representa a variável de entrada do bloco e a variável *Q* representa a variável de saída do bloco. A *logica_entrada* é normalmente representada por uma lógica de controle para ativação com contatos e *variavel_saida* representa, normalmente, uma bobina do programa LD.

O processo FIACRE do ciclo de execução com blocos funcionais nos *rungs*, segue o mesmo padrão do processo FIACRE do ciclo de

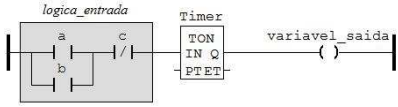


Figura 4.19: Bloco funcional em um rung do LD.

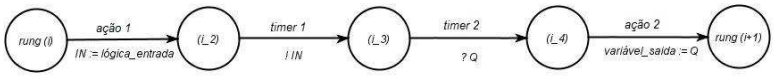


Figura 4.20: Comportamento do bloco funcional no ciclo de execução em rungs do LD.

execução só com elementos básicos. Algumas modificações são, entretanto, necessárias. Uma delas corresponde a necessidade de uma sincronização dos dados do bloco funcional com o ciclo de execução, a segunda é a atribuição dos dados dos blocos funcionais nas variáveis do programa nos *rungs*. No processo FIACRE do ciclo de execução, a atualização dos dados dos blocos funcionais é expressa pelo envio da variável que representa a entrada do bloco funcional (*!IN*), pela transição *timer1*, após a transição de atribuição da lógica de entrada na variável de entrada do bloco (*IN*). O recebimento da variável que representa a saída do bloco (*?Q*), é expressa através da transição *timer2*, antes da atribuição da variável do bloco na variável de saída (*Q*). As ações são definidas no componente com o intervalo *[0,0]*. De acordo com a sintaxe FIACRE, esta parte do processo ciclo de execução, é escrita da seguinte forma:

```
process ciclo_execucao ...
.
.
.
from rung1
  acao1; IN:= logica_entrada;
    to rung1_2
from rung1_2
  timer1 !IN;
    to rung1_3
from rung1_3
```

```
    timer2 ?Q;
      to rung1_4
  from rung1_4
    acao2; variavel_saida:= Q;
      to rung2
  .
  .
  .
```

4.1.6 Modelos dos blocos funcionais

Cada bloco funcional é representado em FIACRE por um processo. A seguir será apresentado o modelo do comportamento do bloco TON, seguido da representação em FIACRE. A representação de outros blocos funcionais com tempo (TOF e CTU) pode ser tratada de forma similar, também apresentada nesta seção.

De acordo com o funcionamento do bloco TON, explicado no Capítulo 2, para representar o comportamento do temporizador TON são necessários três estados. O primeiro é o estado *idle* que está associado à transição que recebe o valor da variável de entrada *IN* do processo ciclo de execução e testa se está ativa (valor booleano *true*). Senão estiver ativa, permanece no mesmo estado e mantém a saída Q desativada. O segundo estado é o *running* que é atingido quando a condição da entrada *IN* estiver ativa. A partir deste estado, é possível zerar o temporizador, voltando para o estado *idle*, caso a entrada *IN* seja desativada antes do tempo. A outra opção, a partir deste estado, consiste em contar o tempo até o valor determinado na variável PT(representado pela transição temporizada *timeout*). Quando este é atingido, através da transição *timeout*, a saída Q é ativada, ocorrendo a mudança para o estado chamado *timeout*. No estado *timeout*, é possível desativar a saída Q do temporizador caso a entrada *IN* for desativada, para voltar ao estado inicial. Todos os estados possuem um *loop* com a transição de atualização da variável Q, pelo envio de mensagem para o processo ciclo de execução. A modelagem do bloco TON foi baseada no modelo apresentado no trabalho de Mokadem [32], com a diferença que

na modelagem deste trabalho não é considerado o compartilhamento de memória entre as variáveis, precisando sempre ocorrer uma transição para atualização das variáveis, de acordo com as limitações da linguagem FIACRE.

Esse processo TON é executado, em paralelo ao processo ciclo de execução, atualizando os valores de entrada e saída do bloco por intermédio da etapa de leitura e atualização dos dados do bloco temporizador. A Fig. 4.21 ilustra o modelo do comportamento do processo bloco temporizador TON, em que os símbolos *'?'* (recebimento de mensagem) e *'!'* (envio de mensagem) representam a sincronização com o processo ciclo de execução.

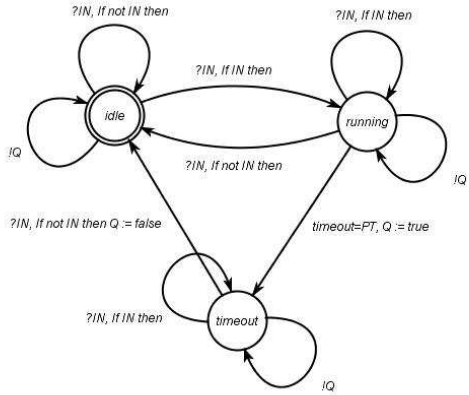


Figura 4.21: Comportamento do processo bloco temporizador TON.

O processo FIACRE, para o caso do bloco temporizador TON, da mesma forma que no modelo anterior, são necessários três estados. O primeiro é o estado *idle* que está associado a um comando *select*, com uma transição chamada *timer1* (para a comunicação com o processo do ciclo de execução) que recebe a mensagem com o valor de *IN* e com uma transição *timer2* que envia uma mensagem para o ciclo de execução com o valor de *Q* e permanece no mesmo estado, por intermédio da instrução

loop. O segundo estado é o *running* que está associado a um comando *select*, que tem como opções uma transição com a condição da entrada ativa, e permite também, a partir do mesmo estado, a transição com as opções de zerar o temporizador desativando a saída Q caso a entrada IN seja desativada, ou de permanecer no estado pela instrução *loop* (usada para não resetar a temporização), ou ainda a ativação do disparo do tempo, através da transição temporizada *timeout*, para depois ativar a saída. Este estado também possui a opção de enviar uma mensagem através da transição *timer2* com o valor de Q. Por fim, é o estado *timeout*, no final do processo, com a condição para desativar a saída Q do temporizador caso a entrada IN for desativada, para que volte ao estado inicial. A atualização dos valores de entrada e saída do bloco ocorre, respectivamente, pelas ações *timer1 e 2* na etapa de sincronização dos dados entre os processos ciclo de execução e bloco funcional. A transição *timeout* associada ao segundo estado (*running*) é temporizada representando a contagem do tempo (a especificação do tempo desta transição equivale ao valor da variável PT do modelo apresentado anteriormente) e é especificada pelo *port* no componente principal, responsável também pela declaração dos processos para que ocorra a comunicação. No caso das ações *timer 1 e 2* a transição é instantânea, e também são especificadas no componente. De acordo com a sintaxe da linguagem FIACRE, o processo TON pode ser representado da seguinte forma:

```
process ton ...
.
.
.
from idle
  select
    timer1 ?IN; if IN then to running
                else loop end
    [] timer2 !Q; loop
  end
from running
  select
    timer1 ?IN; if not IN then Q:= false; to idle
                else loop end
```

```
    [] timeout; Q:= true; to timeout
    [] timer2 !Q; loop
  end
from timeout
  select
    timer1 ?IN; if not IN then Q:= false; to idle
                else loop end
    [] timer2 !Q; loop
  end
```

Para o bloco temporizador TOF também são necessários três estados. O primeiro é o estado *idle* que está associado à transição que recebe o valor da variável de entrada *IN* e testa se está ativa. Senão estiver ativa, permanece no mesmo estado e mantém a saída *Q* desativada. Se *IN* estiver ativo, ativa a saída *Q*, e vai para o estado *active*. O segundo estado *active* está associado com a condição da entrada *IN* estar ativa, com as opções de manter no mesmo estado, caso a entrada *IN* permaneça ativa, ou ir para o estado *running*, caso a entrada seja desativada. O estado *running* tem as opções de zerar o temporizador, voltando para o estado *active*, caso a entrada *IN* volte a ser ativada antes do tempo, ou contar o tempo até o valor determinado na variável *PT*, através da transição temporizada *timeout*, para depois desativar a saída *Q* e seguir para o estado *idle* novamente. Todos os estados possuem um *loop* com a transição de atualização da variável *Q*, pelo envio de mensagem para o processo ciclo de execução. A Fig. 4.22 representa o comportamento do processo bloco temporizador TOF.

Na linguagem FIACRE, para o bloco temporizador TOF, da mesma forma que no modelo, também são necessários três estados. O primeiro é o estado *idle* que está associado a uma instrução *select*, com uma transição *timer1* que recebe a mensagem com o valor de *IN* e testa se está ativo e com uma transição *timer2* que envia a mensagem com o valor de *Q*. O estado *active* testa novamente a entrada *IN*, se for desativada vai para o estado *running*, senão permanece no mesmo estado. O estado *running* está associado ao comando *select*, com uma transição representando a condição de testar a entrada, com uma transição com

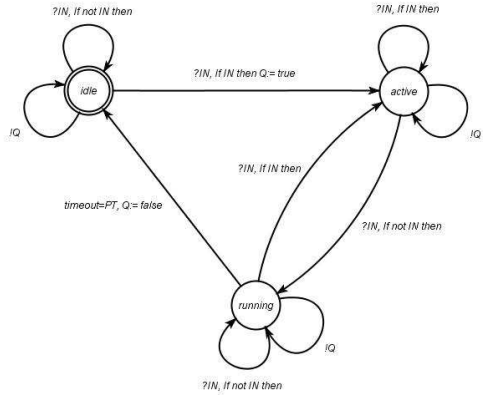


Figura 4.22: Comportamento do processo bloco temporizador TOF.

as opções de zerar o temporizador caso a entrada *IN* seja ativada, voltando para o estado anterior, ou permanecendo no estado através da instrução *loop*, ou ainda, uma transição que permite ativar o disparo do tempo, através da transição temporizada *timeout*, para depois desativar a saída, seguindo para o estado *idle*, e também uma transição representando a possibilidade do envio do valor de Q através da transição *timer2*. Da mesma forma que no processo TON, a transição *timeout* é temporizada e representa a contagem do tempo (a especificação do tempo desta transição equivale ao valor da variável PT do modelo de funcionamento). De acordo com a sintaxe, o comportamento processo TOF é representado da seguinte forma:

```
process tof ...
.
.
.
from idle
  select
    timer1 ?IN; if IN then Q:= true; to active
               else loop end
    [] timer2 !Q; loop
  end
```

```
from active
  select
    timer1 ?IN; if not IN then to running
                else loop end
    [] timer2 !Q; loop
  end
from running
  select
    timer1 ?IN; if IN then  to active
                else loop end
    [] timeout; Q:= false; to idle
    [] timer2 !Q; loop
  end
```

De acordo com a explicação do Capítulo 2 sobre o funcionamento dos blocos, para o bloco contador CTU também são necessários três estados. O primeiro é o estado *idle* que está associado a condição de teste da variável CTU (variável de entrada do bloco) e $CV<PV$. Se a condição for verdadeira, vai para o estado *counting* e incrementa a variável CV (variável contadora), senão permanece no mesmo estado e mantém a saída Q desativada. Na variável PV está armazenado o valor limite para a contagem. No estado *counting*, se $CV>=PV$ então ativa a saída Q e vai para o estado *active*, caso contrário incrementa a variável CV a cada ativação de CTU, e permanece no mesmo estado. O estado *counting* também pode ser resetado pela variável R, voltando para o estado *idle*, e zerando a variável CV. O estado *active* é mantido até ocorrer um reset (R), atualizando o valor de Q para falso e voltando para o estado *idle* novamente. Da mesma forma que os modelos anteriores, todos os estados possuem um *loop* com a transição de atualização da variável Q para o ciclo de execução. A Fig. 4.23 representa o comportamento do processo bloco contador CTU.

Em FIACRE, para o bloco contador CTU, também são necessários três estados. O primeiro é o estado *idle* que está associado às transições *count1* que recebe a mensagem com o valor de CTU e testa se está ativo, se sim, incrementa a variável CV e vai para o estado *counting* e *count2* que envia o valor de Q para o processo ciclo de execução. O estado

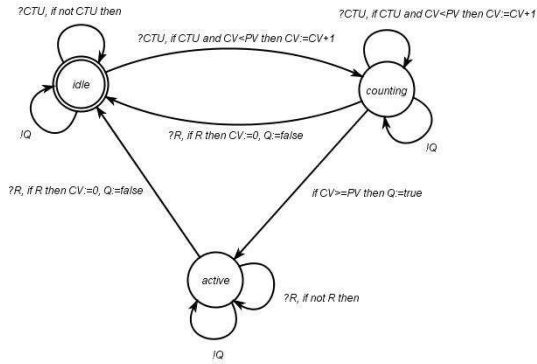


Figura 4.23: Comportamento do processo bloco contador CTU.

counting está também associado a um comando *select*, com a transição de zerar o contador, caso a variável R seja ativada, voltando para o estado anterior, ou permanecendo no mesmo estado, ou a transição se a condição de CTU for verdadeira, de incrementar a variável CV, até o valor estipulado em PV, e finalmente a transição com o envio de Q. O estado *active* testa novamente a variável R, se for ativada vai para o estado *idle* e zera a variável CV, senão permanece no mesmo estado. A opção de envio do valor da variável Q permanece válido no *select*. De acordo com a sintaxe, o comportamento em FIACRE pode ser representado da seguinte forma:

```
process ctu ...
.
.
.
from idle
  select
    count1 ?CTU; if CTU and CV<PV then CV:= CV+1; to counting
                 else loop end
    [] count2 !Q; loop
  end
from counting
  select
    reset ?R; if R then CV:=0; to idle
              else loop end
```

```
    [] count1 ?CTU; if CTU and CV<PV then CV:= CV+1; loop
                    else Q:= true; to active end
    [] count2 !Q; loop
  end
from active
  select
    reset ?R; if R then CV:=0; Q:= false; to idle
              else loop end
    [] count2 !Q; loop
  end
```

4.2 Exemplo com elementos básicos e bloco TON

Para facilitar o entendimento das construções em FIACRE, a Fig. 4.24 mostra um exemplo de programa com os elementos básicos (contato NA, contato NF e bobina NA) e um bloco funcional (temporizador TON). O contato *I1* ativa a bobina *O1*, que se mantém ativada por também estar ligada em paralelo com o contato *O1*. A variável *O1* é a responsável, também, pelo acionamento do bloco temporizador TON, que após 5 segundos (tempo determinado em PT), ativa a bobina *O2*. O contato *I2* funciona como um *reset* do programa. A seguir é apresentada o código em FIACRE correspondente ao exemplo da Fig. 4.24.

O programa possui o processo do ciclo de execução, com a sincronização dos dados dividida em duas partes (envio da variável *IN* e recebimento da variável Q), execução do programa, com os elementos simples e atualização de entrada e saída do bloco, e reinício do ciclo. Neste exemplo não está representado a etapa de leitura e escrita dos dados do ambiente externo, podendo ser simulado através dos valores iniciais das variáveis (atribuindo os valores iniciais na chamada dos processos no componente principal). As representações do ambiente externos serão apresentadas no próximo Capítulo. No cabeçalho do processo ocorre a declaração das transições e variáveis utilizadas. O processo do bloco TON possui os elementos de acordo com a descrição

apresentada nesta seção, e se comunica com o ciclo de execução através das transições que enviam e recebem dados (*timer 1 e 2*). O componente principal (*main*) possui a declaração dos tempos das transições (ações), representadas pelos *ports*, a composição dos processos, é feita pelo comando *par*, no qual permite a comunicação entre os processos, juntamente com a definição do valor inicial das variáveis do programa. O código em FIACRE para o exemplo é o seguinte:

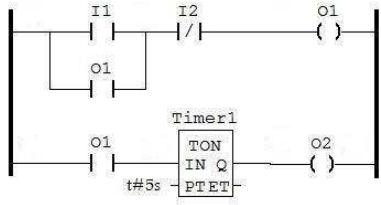


Figura 4.24: Exemplo de programa em LD com elementos básicos e bloco temporizador.

```
/* Cabeçalho do processo com as transições e variáveis */

process ciclo_execucao [timer1, timer2: bool, rung_simples, rung_timer1,
rung_timer2, restart: none](I1, I2, O1, O2, IN, Q: bool) is

/* Declaração dos estados */

states rung1, rung2, rung2_2, rung2_3, rung2_4, fim

/* Definição do estado inicial */

init to rung1

/* Início do programa */

from rung1
  rung_simples; O1:= (I1 or O1) and not I2;
      to rung2
from rung2
  rung_timer1; IN:= O1;
```

```
      to rung2_2
from rung2_2
  timer1 !IN;
      to rung2_3
from rung2_3
  timer2 ?Q;
      to rung2_4
from rung2_4
  rung_timer2; O2:= Q;
      to fim
from fim
  restart;
      to rung1

/* Processo do temporizador TON */

process ton [timer1, timer2: bool, timeout: none](IN, Q: bool) is

states idle, running, timeout
init to idle

from idle
    select
      timer1 ?IN; if IN then to running
                  else loop end
      [] timer2 !Q; loop
    end
from running
  select
    timer1 ?IN; if not IN then Q:= false; to idle
                else loop end
    [] timeout; Q:= true; to timeout
    [] timer2 !Q; loop
  end
from timeout
  select
    timer1 ?IN; if not IN then Q:= false; to idle
                else loop end
    [] timer2 !Q; loop
  end

/* Componente principal do sistema */
```

```
component main is

/* Declaração dos intervalos de tempo das transições */

  port
    timer1:       bool in [0,0],
    timer2:       bool in [0,0],
    rung_simples: none in [0,0],
    rung_timer1:  none in [0,0],
    rung_timer2:  none in [0,0],
    restart:      none in [1,1],
    timeout:      none in [5,5]

/* Operação de comunicação paralela entre os processos e
   definição dos valores iniciais das variáveis do sistema */

  par
    timer1, timer2, rung_simples, rung_timer1, rung_timer2, restart ->
    ciclo_execucao [timer1, timer2, rung_simples, rung_timer1,
    rung_timer2, restart] (true, false, false, false, false, false)
    || timer, timer2, timeout -> ton [timer, timer2, timeout]
    (false, false)
  end
main
```

De forma geral, o programa de CLP, representado em FIACRE, vai conter: o componente principal (com a declaração dos tempos das transições de todos os processos, a chamada dos processos que compõem o programa com os valores iniciais das variáveis, juntamente com a operação para fazer a comunicação paralela entre os processos), o processo do ciclo de execução, os processos dos blocos funcionais com tempo e os processos (ou componente) representando o ambiente externo.

4.3 Conclusão do capítulo

Foram apresentadas, neste capítulo, as modelagens do comportamento de um programa em LD de CLP, que constituem a base para a transformação para a linguagem FIACRE. Foram descritas as regras

gerais de tradução de programas em LD para FIACRE, de acordo com a modelagem apresentada, juntamente com as construções em FIACRE dos principais elementos que compõem um programa LD. Um exemplo, que facilita o entendimento das equivalências entre as linguagens, também foi apresentado, utilizando elementos básicos e o bloco temporizador TON.

A partir do programa de CLP, na linguagem LD, transformado para FIACRE, é possível utilizar os compiladores já implementados (FIACRE-TTS e FIACRE-LOTOS) para utilização nas ferramentas de verificação formal (TINA e CADP). O próximo capítulo irá tratar a verificação de propriedades aplicada em programas de LD transformados nas linguagens correspondentes, de acordo com a ferramenta de verificação.

Capítulo 5

Verificação de propriedades para programas de LD

De acordo com o que foi apresentado no Capítulo 3, neste trabalho serão aplicadas duas técnicas de verificação formal, por *model-checking* e por equivalências de modelos, utilizadas no ambiente Topcased. Ambas necessitam de linguagens (FIACRE, TTS, LOTOS) e ferramentas específicas (TINA, CADP) conforme foram apresentadas e que serão aplicadas neste Capítulo pelas duas cadeias de verificação. Antes é necessário a representação da modelagem da planta, juntamente com as propriedades a serem verificadas.

Neste capítulo é apresentada a verificação de propriedades, aplicada a exemplos de programas em LD. São descritas as propriedades, gerais e específicas, de programas de CLP que serão utilizadas nas cadeias de verificação. É feita também a modelagem do ambiente externo, necessária para a verificação de cada um dos exemplos. Neste capítulo também ocorre a descrição dos exemplos com a tradução das propriedades, representação dos modelos e a aplicação nas cadeias de verificação

formal.

5.1 Propriedades para a verificação

Nesta seção é apresentada a modelagem das propriedades do LD, que é dividida basicamente em duas partes. A primeira são as propriedades gerais, as quais podem ser aplicadas a todos os exemplos de programas de CLPs. A segunda são as propriedades específicas, relacionadas ao modelo, e que dependem da aplicação.

5.1.1 Propriedades gerais

As propriedades gerais, ou genéricas, podem ser aplicadas a todos os modelos de programas de CLP, mais especificamente aqueles escritos em LD. Nesta seção são apresentadas algumas propriedades com suas descrições e formas de especificar. Por exemplo, verificar a ocorrência de *deadlocks* é sempre importante. O *deadlock* ocorre quando mais de um processo requer um determinado recurso ao mesmo tempo, ficando impedidos de continuar suas execuções, ou seja, ficam bloqueados.

A propriedade de vivacidade (*liveness*) significa ser possível, a partir de qualquer um dos estados alcançáveis do sistema, executar todas as suas ações. Esta propriedade é muito importante na análise de sistemas, pois quando verificada, indica a inexistência de bloqueios no sistema modelado. No caso da vivacidade, é importante considerar que um evento desejado sempre ocorrerá, independente da ocorrência de outros eventos associados ao modelo, ou seja, indica que determinados estados do sistema tem de ser atingidos.

Outra propriedade importante é a alcançabilidade (*reachability*). A propriedade de alcançabilidade ocorre se existir uma sequência finita de ações que a conduza a partir de um determinado estado. Se todas as ações alcançáveis forem decorrentes deste estado, o sistema é dito alcançável. A lógica temporal deve ser feita para que uma determinada situação, que se deseja verificar, possa ser atingida. Esta propriedade

pode ser expressa de forma contrária também, ou seja, que um determinado evento nunca ocorre.

A propriedade de justiça (*fairness*) indica que, sob certas condições, um evento deve ocorrer um número infinito de vezes. Por exemplo, um transmissor que tenta enviar suas mensagens um número infinito de vezes consegue sucesso na transmissão em um número infinito de vezes. Sob a forma de lógica pode ser representada, por exemplo, da seguinte forma: $[]\Diamond$ *msg_enviada* $\Rightarrow$ $[]\Diamond$ *msg_recebida*.

As questões de segurança (*safety*) devem ser levadas em consideração, e são propriedades que são verificadas para todas as execuções e em todos os estados. Uma propriedade de segurança especifica que um determinado evento nunca deve ocorrer, de acordo com certas condições, por exemplo, uma máquina só deve funcionar caso as condições tenham sido cumpridas [17].

5.1.2 Propriedades específicas

As propriedades específicas são relacionadas diretamente aos modelos dos sistemas a serem verificados [35]. Existem diversas formas de se especificar e extrair propriedades a partir das especificações dos programas escritos em LD do CLP. Uma maneira mais informal é através da descrição literal das especificações do sistema, em que é feita a formulação verbal do problema. A escrita das especificações é feita a partir do conhecimento do sistema, por meio de uma especificação natural estruturada. É necessária também a elaboração da tabela de correspondência lógica, para definir qual o significado dos estados lógicos.

Uma forma de especificar os requisitos dos sistemas de LD, com maiores detalhes, é através da matriz de causa e efeito, facilitando a extração de propriedades através de lógicas, por exemplo, esta é a forma adotada pelo setor do petróleo (em particular, pela Petrobras). A matriz de causa e efeito trata-se de uma tabela em que se estabelecem as relações de acionamento do sistema em sequência. Na primeira linha da tabela listam-se os sinais que são as causas para os acionamentos, e na primeira coluna listam-se os consequentes efeitos nos sinais do sistema.

Outra forma de descrever as especificações (propriedades do sistema) de um programa mais simples e sequencial é o diagrama trajeto-passo (adotado geralmente pelas indústrias que trabalham com equipamentos pneumáticos). Para projetos maiores, e com paralelismo, pode ser também utilizada a linguagem Grafcet, ou SFC, para auxiliar na descrição do comportamento do sistema. O diagrama trajeto-passo torna possível a visualização global dos movimentos e interações do sistema e suas relações de dependência, e é bastante utilizado em aplicações pneumáticas. Por meio do diagrama trajeto-passo é possível extrair algumas propriedades específicas do sistema, como por exemplo, a ordem de precedência de ativação de cilindros de um sistema pneumático, como será visto no exemplo deste Capítulo. Outro requisito importante, que este diagrama permite constatar, é se ocorre a colisão física dos elementos do sistema, ou seja, no caso dos cilindros, se estão estendidos ao mesmo tempo, pois pode não ser viável este acontecimento fisicamente.

Para os exemplos que serão apresentados, é importante verificar também propriedades específicas relacionadas com a segurança física dos sistemas. Como por exemplo, em um sistema industrial, se os sinais não ocorrem ao mesmo tempo, podendo ocasionar defeito em um motor, ou a precedência da ordem de acionamento de motores, para que o sistema funcione corretamente sem riscos de problemas, ou ainda se os sensores ativam, realmente, os motores certos.

5.2 Modelagem da comunicação com o ambiente externo

No Capítulo 4 foram apresentadas as modelagens do comportamento de um programa em LD necessárias para a transformação das linguagens da cadeia de verificação. Neste Capítulo são apresentadas as modelagens do ambiente externo do LD necessárias para a aplicação da verificação formal. Para isso, é preciso caracterizar a planta que o programa do CLP deve controlar, por meio da modelagem dos elementos

do sistema. A modelagem do ambiente externo de um CLP depende da aplicação. Neste capítulo são tratados os elementos que serão utilizados na aplicação dos exemplos deste trabalho. Primeiramente são modelados os processos *glue*, responsáveis pela interface de comunicação com a planta. Depois, os elementos que representam a planta, como por exemplo, os cilindros, a ventosa, o motor simples e o motor bidirecional. As plantas escolhidas como exemplos neste trabalho são um sistema de automação pneumática e um sistema para um misturador industrial que serão apresentados nas próximas seções.

De acordo com a modelagem do ciclo de execução, apresentada no capítulo anterior, está prevista a leitura dos sinais oriundos do ambiente externo e a escrita dos sinais neste ambiente. Estas etapas de comunicação permitem a interação do processo representando o ambiente com o envio e recebimento de mensagens com os valores das variáveis para o processo ciclo de execução do programa em LD. A representação do ambiente é dividida de acordo com o número de elementos do sistema (componentes da planta), ou seja, vai existir um processo para cada componente.

O *glue*, que representa a interface de comunicação com o ambiente externo, é divido em dois processos representando os dados de entrada (por exemplo, sensores) e os dados de saída (por exemplo, atuadores). A Fig. 5.1 representa os modelos do comportamento do *glue* que fazem a comunicação do processo ciclo de execução com os processos do ambiente externo, em que o programa principal (processo ciclo de execução) exercerá o controle sobre esses processos de comunicação. O processo representando os sensores (*inputsGlue*) recebe o dado de entrada do ambiente externo (transição *sensor*, transmitindo o dado de entrada) e envia para o ciclo de execução (transição *leitura*, transmitindo o dado de entrada). Em FIACRE, o processo *glue* representando os sensores é apresentado da seguinte forma:

```
process inputsGlue ...
.
.
.
```

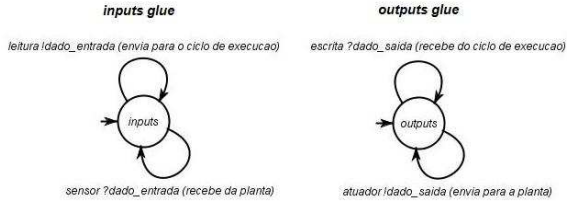


Figura 5.1: Modelo dos processos glue de entrada e saída

```
from input
        select
                leitura !dado_entrada; to input
                [] sensor ?dado_entrada; to input
        end
```

O processo *glue*, que representa a comunicação do dado de saída, tem comportamento similar ao processo dos sensores, modificando apenas o nome das transições e os dados. O processo representando os atuadores recebe o dado de saída do ciclo de execução (através da transição *'escrita'*, transmitindo o dado de saída) e envia para o ambiente externo (através da transição *'atuador'*, transmitindo o dado de saída). Em FIACRE, o processo *glue* representando os atuadores é apresentado da seguinte forma:

```
process outputsGlue ...
.
.
.
from output
        select
                escrita ?dado_saida; to output
                [] atuador !saida; to output
        end
```

5.3 Exemplo de verificação de um sistema de automação pneumática controlado por CLP

Nesta seção será apresentado um exemplo ilustrativo de verificação formal de programas de CLP escritos em LD que controlam uma planta. A modelagem da comunicação com o ambiente externo, apresentada na seção anterior, será utilizada também neste exemplo, nos quais a aplicação é constituída de um programa em LD para o controle e da representação da planta. Também serão apresentados os aspectos da transformação das propriedades e a aplicação destas no processo de verificação.

5.3.1 Descrição do sistema de automação pneumática

O exemplo desta seção foi desenvolvido pelo grupo de pesquisa do Laboratório de Sistemas Hidráulicos e Pneumáticos da UFSC (LASHIP) [5]. O sistema apresentado (Módulo 3) faz parte de um conjunto de módulos de controle de uma bancada didática de um Sistema de Automação Pneumática (SAP). Este módulo é destinado à movimentação de caixas de uma posição (geralmente próxima aos módulos de alimentação) para outra, localizada na outra extremidade do cilindro sem haste (*3A1*). A caixa é pega próxima à mesa (cilindro com guias *3A2* avançado) pela ventosa (*3A3*) e solta na outra ponta do cilindro *3A1* em uma posição elevada (cilindro *3A2* recuado). O módulo, quando chamado, deve pegar a caixa, movimentá-la, deixando-a na posição programada e retornar para a posição inicial. A chamada do módulo deve ser executada pelo comando *Mod3Ini*. O fim da execução do programa do Módulo 3 é indicado pela variável *Mod3Fim*. O módulo é composto por um cilindro de duplo acionamento sem haste (*3A1*) com sensores de fim de curso (*3S1 e 3S2*), um cilindro de duplo acionamento com guias (*3A2*) com sensores de fim de curso (*3S3 e 3S4*) e uma ventosa (*3A3*), que possui as variáveis *T1 e T2*. A Fig. 5.2 ilustra fisicamente como é

composto o Módulo 3 do sistema de automação pneumática (SAP).

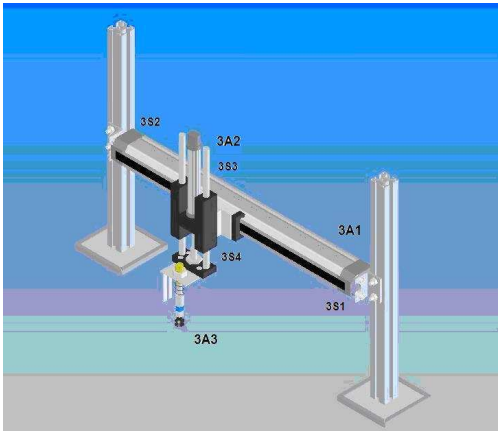


Figura 5.2: Representação física do módulo 3 do SAP [5].

O diagrama trajeto-passo mostra o comportamento desejado dos diversos elementos da planta (cilindros, ventosa), sua relação entre si, com os sinais de comando oriundos do CLP e também com os sinais dos sensores. O diagrama trajeto-passo neste exemplo é dividido em três partes, uma para cada elemento, sendo que cada nível corresponde a um estado deste elemento. Os sinais entre cada uma das partes do diagrama representam as relações de precedência exigidas entre cada elemento e suas relações com os comandos do CLP. A Fig. 5.3 mostra o diagrama trajeto-passo do módulo 3 do SAP.

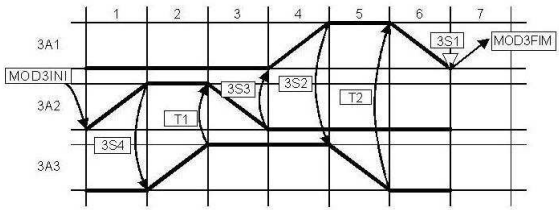


Figura 5.3: Diagrama trajeto-passo do módulo 3 do SAP [5].

O programa em LD para o controle do SAP é apresentado na

Fig. 5.4. Este programa, proposto em [5], controla a planta descrita anteriormente. Um dos objetivos deste trabalho consiste em mostrar que o sistema composto do CLP (com o programa proposto) e da planta atende as propriedades descritas no diagrama trajeto-passo.

5.3.2 Modelagem dos cilindros e ventosa do SAP

A modelagem dos elementos utilizados no exemplo do sistema de automação pneumática consiste na representação de dois cilindros (*3A1 e 3A2*) e de uma ventosa (*3A3*). Os cilindros possuem quatro estados: *recuado, avançando, avançado e recuando*. O cilindro *3A1* quando recebe o sinal *3S3* do programa de controle começa a avançar (estado *avançando*). Quando ocorre o sinal do sensor *3S2*, chega ao estado *avançado*. Quando recebe o sinal *T2*, vai para o estado *recuando*, após envia o sinal do sensor *Mod3Fim* e volta para o estado inicial *recuado*. O cilindro *3A2* funciona de forma semelhante, possui os mesmos estados do cilindro *3A1*, mudando apenas os sinais de controle. O segundo cilindro recebe o sinal *Mod3Ini* para começar a avançar, depois envia o sinal *3S4* e vai para o estado *avançado*. Recebe o sinal por meio de *T1* para começar a recuar, depois envia o sinal *3S3* e vai para o estado *recuado* novamente. O comportamento da ventosa *3A3* é um pouco diferente, pois possui os estados *desligada, acionando, acionada e desligando*. A ventosa recebe o sinal *3S4* para acionar, depois envia o sinal *T1* e vai para o estado *acionada*. Recebe o sinal *3S2* para desligar, depois envia o sinal *T2* e vai para o estado *desligada* novamente. As Figuras 5.5, 5.6 e 5.7 representam as modelagens dos cilindros *3A1, 3A2* e da ventosa *3A3*, respectivamente.

A partir da modelagem do comportamento desses elementos é possível descrever em FIACRE os processos que representam os cilindros e a ventosa com os quais programa do CLP vai interagir. A partir desta modelagem da planta controlada pelo CLP, será possível fazer a verificação utilizando os métodos e ferramentas citados anteriormente. As especificações em FIACRE da planta física, da mesma forma que os modelos, foram divididas em três processos: dois representando os ci-

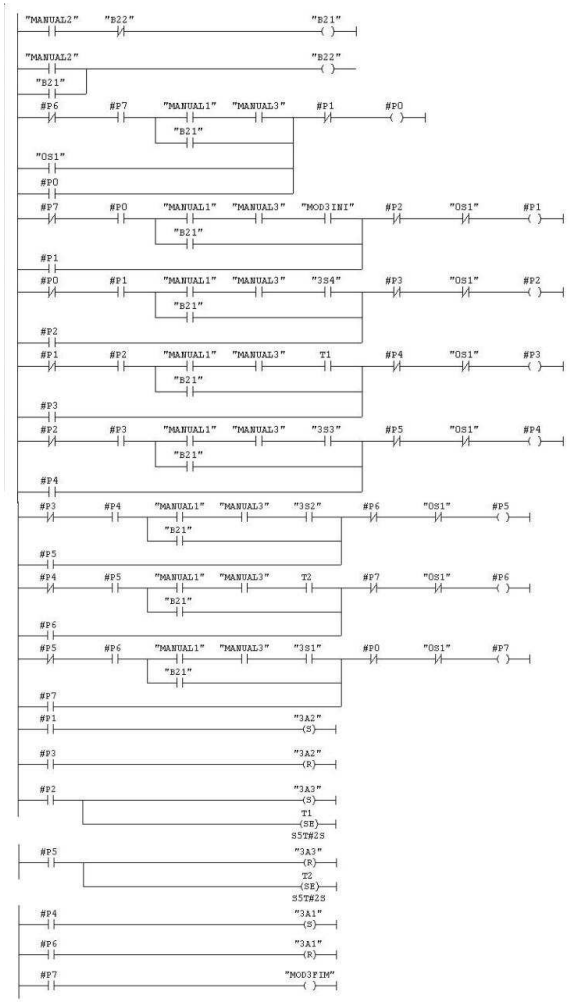


Figura 5.4: Programa em LD para o módulo 3 do SAP [5].

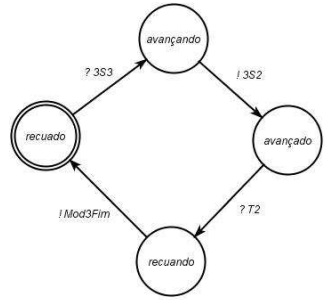


Figura 5.5: Comportamento do cilindro 3A1.

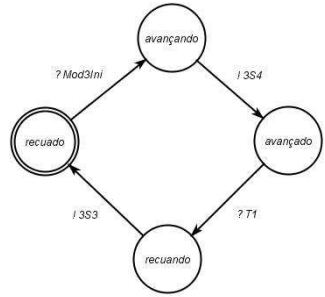


Figura 5.6: Comportamento do cilindro 3A2.

lindros (*3A1 e 3A2*) e um representando a ventosa (*3A3*). Todas as variáveis que começavam com número foram invertidas e começam com letra em FIACRE, pois os nomes de variáveis em FIACRE não podem iniciar com número. Para a criação de cada processo, foi utilizada a máquina de estados apresentada anteriormente, ou seja, cada estado do modelo corresponde a um estado do processo que representa o cilindro e os sinais representam as transições que permitem a troca de estados. O cilindro 2 (*3A2*) e a ventosa (*3A3*) tem comportamento similar, mudando apenas os sinais de controle. O processo em FIACRE do cilindro 1 (*3A1*) é o seguinte:

```
process planta_cilindro1 [sinal3S3, leitura_3S2, sinalT2, leitura_Mod3Fim
: bool] (S3_3, S3_2, T2, Mod3Fim: bool) is
```

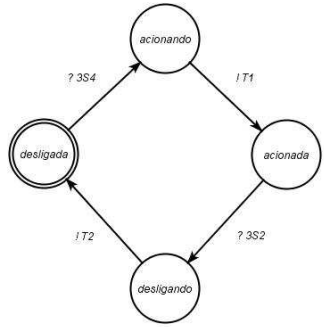


Figura 5.7: Comportamento do cilindro 3A3.

```
states cilindro1_recuado, cilindro1_avancado, avancando_cilindro1,
recuando_cilindro1
init to cilindro1_recuado

from cilindro1_recuado sinal3S3 ?S3_3;
  to avancando_cilindro1
from avancando_cilindro1 S3_2:= true; leitura_3S2 !S3_2;
  to cilindro1_avancado
from cilindro1_avancado sinalT2 ?T2;
  to recuando_cilindro1
from recuando_cilindro1 Mod3Fim:= true; leitura_Mod3Fim !Mod3Fim;
  to cilindro1_recuado
```

O modelo completo em FIACRE com todos os elementos da planta, com o programa de controle do CLP e o modelo de comunicação com a planta do SAP estão no Apêndice B1, juntamente com o componente principal, que é responsável pela comunicação entre todos os processos em FIACRE.

5.3.3 Modelagem e verificação das propriedades do SAP

De acordo com a abordagem de verificação, é preciso traduzir as propriedades para o formalismo adequado. As propriedades, gerais e específicas, são obtidas segundo as sugestões apresentadas nas seções

anteriores. Para as duas abordagens de verificação, utilizadas neste trabalho, é necessário no caso da verificação por *model-checking*, transformar as propriedades desejadas em formulas da lógica temporal LTL, e no caso da verificação por equivalência, de transformá-las em especificações em FIACRE que poderá ser posteriormente transformado em LOTOS.

5.3.3.1 Verificação por model-checking do SAP

Para a abordagem lógica, utilizando a verificação por *model-checking*, é necessário traduzir as especificações das propriedades desejadas do sistema para uma lógica temporal para poder verificá-las no programa em LD do CLP, em conjunto com a planta, traduzido para FIACRE, e posteriormente, para TTS. Como a ferramenta utilizada para o *model-checking* é o TINA, a entrada deve ser especificada na lógica LTL (de acordo com a Fig. 3.3 apresentada no Capítulo 3).

Neste trabalho, o sistema global é representado na forma de um TTS, resultado da tradução via linguagem intermediária FIACRE da composição do programa em LD do CLP com a representação da planta, e as propriedades são expressas na forma de fórmulas da lógica temporal LTL. Por sua vez, a ferramenta SELT/TINA permite verificar por *model-checking*, as propriedades em LTL sobre o sistema descrito em TTS. Por intermédio do programa em LD do SAP (editado no PLCOpen Editor) é gerado o arquivo em FIACRE (apresentado no Apêndice B.1), que serve de entrada para o compilador FIACRE/TTS. Por meio do arquivo TTS gerado, é utilizada a ferramenta SELT/TINA juntamente com as fórmulas em LTL. O Apêndice B.2 mostra o resultado completo das verificações do exemplo módulo 3 do SAP utilizando o verificador SELT/TINA.

Para o exemplo do Módulo 3 do SAP foram seguidos os seguintes passos para especificar as propriedades. Primeiramente, é feita a descrição detalhada do problema, com as especificações e requisitos do sistema. Depois, para facilitar o entendimento do funcionamento do sistema, pode ser feita a elaboração de uma tabela de correspondência

lógica, para definir qual o significado dos estados lógicos das entradas e saídas. Acrescentando-se a isto, o diagrama-trajeto passo, que caracteriza o estado dos atuadores, assim como as conseqüências oriundas dos sinais de controle do sistema. De posse do diagrama-trajeto passo, juntamente com a modelagem, é possível extrair as propriedades do sistema. As relações de dependência e a ordem temporal obtidas a partir do diagrama podem ser facilmente transformadas em fórmulas da lógica temporal utilizada. Para a modelagem o diagrama trajeto-passo é dividido em três partes, uma para cada elemento, sendo que cada nível corresponde a um estado deste elemento. Os sinais entre cada uma das partes do diagrama representam as relações de precedência exigidas entre cada elemento e suas relações com o sistema de controle.

De acordo com o que foi tratado na seção sobre propriedades específicas, é necessário a verificação da precedência de ativação dos cilindros e da ventosa, para o sistema do Módulo 3 do SAP, conforme as seguintes propriedades e a sua tradução para a lógica temporal LTL.

Após o avanço do cilindro 2, então a ventosa acionará:

$[]\ (avancando\_cilindro2 \Rightarrow \Diamond\ acionando\_ventosa)$

O resultado verdadeiro ($TRUE$) do verificador SELT/TINA confirma a ordem de precedência assim descrita conforme apresentada no diagrama trajeto-passo.

Após o acionamento da ventosa, então o cilindro 1 avançará:

$[]\ (acionando\_ventosa \Rightarrow \Diamond\ avancando\_cilindro1)$

Da mesma forma que a anterior, o resultado verdadeiro ($TRUE$) gerado confirma a ordem de precedência.

Após o acionamento da ventosa, então o cilindro 2 avançará:

$[]\ (acionando\_ventosa \Rightarrow \Diamond\ avancando\_cilindro2)$

O resultado falso ($FALSE$) do verificador SELT/TINA confirma que esta ordem de ativação não é permitida.

Após o avanço do cilindro 1, então a ventosa acionará:

$[]\ (avancando\_cilindro1 \Rightarrow \Diamond\ acionando\_ventosa)$

Da mesma forma que a anterior, o resultado falso ($FALSE$) gerado confirma que esta ordem também não é permitida.

Conforme diagnosticado, outro requisito importante, é se ocorre a colisão dos elementos do sistema, ou seja, no caso dos cilindros, se estão estendidos ao mesmo tempo, como os seguintes. O sinal - na lógica indica a negação da sentença.

O cilindro 1 e o cilindro 2 nunca estarão avançados ao mesmo tempo:

[] - ($cilindro1\_avancado \wedge cilindro2\_avancado$)

O resultado *TRUE* da verificação permite afirmar que não haverá colisão física dos elementos.

O cilindro 2 e a ventosa nunca estarão ativos ao mesmo tempo:

[] - ($cilindro2\_avancado \wedge ventosa\_acionada$)

O resultado *FALSE* do verificador confirma que a premissa anterior é verdadeira. De acordo com o diagrama trajeto-passo não há problema na ativação do cilindro e da ventosa ao mesmo tempo.

O cilindro 1 e a ventosa nunca estarão ativos ao mesmo tempo:

[] - ($cilindro1\_avancado \wedge ventosa\_acionada$)

Da mesma forma que a anterior, o resultado falso (*FALSE*) gerado confirma que os cilindros e a ventosa podem ser acionados ao mesmo tempo.

Os dois cilindros e a ventosa nunca estarão ativos ao mesmo tempo:

[] - ($cilindro1\_avancado \wedge cilindro2\_avancado \wedge ventosa\_acionada$).

O resultado *TRUE* da verificação confirma que não haverá um instante de tempo em que todos os elementos estejam acionados ao mesmo tempo.

As propriedades apresentadas nesta seção são apenas um exemplo do que é possível e outras propriedades também podem ser verificadas. Estas propriedades garantem a verificação do funcionamento básico do SAP, mas outras situações e modificações podem ser acrescentadas para aumentar ainda mais a confiabilidade do sistema de acordo com as propriedades desejadas. A extração automática das propriedades, e transformação em lógica temporal, é dificultada pelas diversas formas, com diversos graus de rigor na sua representação. A forma mais co-

mum de um usuário descrever estas propriedades é através do uso da linguagem natural com toda as eventuais ambiguidades, contradições e redundâncias que esta comporta. A extração de fórmulas de lógica temporal a partir de frases em linguagem natural pode ser facilitada pelo uso de palavras tais como *sempre, no futuro, infinitas vezes, até* que permitem expressar relações de ordem entre eventos ou estados ou como *então, nunca, e, ou*, entre outras, para expressar proposições lógicas.

5.3.3.2 Verificação por equivalência do SAP

Para a abordagem comportamental, utilizando a verificação por equivalência, é necessário traduzir as especificações das propriedades desejadas do SAP para um código em FIACRE, que servirá para verificar a equivalência com o SAP contendo o programa LD e a planta, traduzido para FIACRE. Estes dois códigos FIACRE são transformados na linguagem LOTOS que serve de entrada para a ferramenta CADP desta cadeia de verificação. Como a ferramenta utilizada para fazer as equivalências é o CADP, é ainda necessária a utilização do compilador de FIACRE para LOTOS, para a entrada na ferramenta (de acordo com a Fig. 3.4 apresentada no Capítulo 3).

De acordo com a descrição do sistema e do diagrama trajeto-passo do módulo 3 do SAP, que mostra o comportamento dos sinais e da atuação dos cilindros e da ventosa, foi possível fazer a modelagem destas propriedades. Cada linha do diagrama representa um elemento do sistema (cilindro *3A1* e *3A2* e ventosa *3A3*) e cada estado do diagrama corresponde a um estado dos cilindros e ventosa (recuado, avançando, avançado e recuando). Os sinais de comunicação (sinais dos sensores e atuadores do sistema) representam as transições entre os estados. Foram criados dois estados simbolizando o ambiente, que representam a entrada do sinal oriundo do módulo anterior e a saída do sinal para o próximo módulo do SAP, que não fazem parte deste exemplo ilustrativo. A Fig. 5.8 representa a modelagem das propriedades (correspondentes ao comportamento desejado) do módulo 3, de acordo com o diagrama trajeto-passo, através de um sistema de transição, para facilitar a visu-

alização dos sinais.

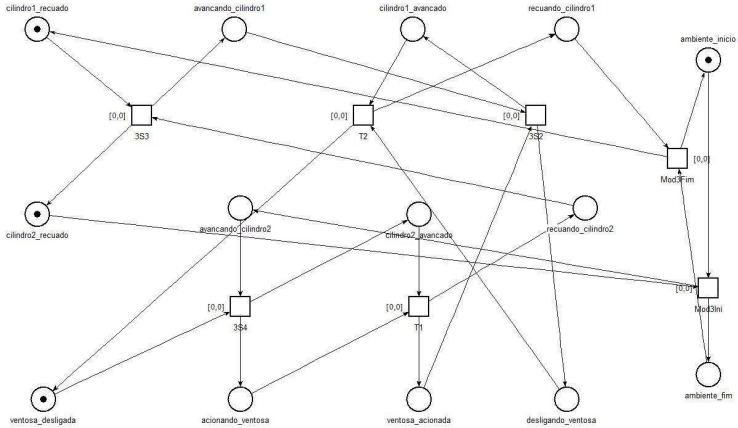


Figura 5.8: Modelagem geral das especificações do comportamento do módulo 3 do SAP.

Para o SAP, de acordo com a modelagem, foi possível descrever as propriedades do sistema por meio da linguagem FIACRE. A finalidade desta etapa é poder observar e comparar se o resultado da transformação para FIACRE do programa descrito em LD, juntamente com a planta, está em algum tipo de equivalência com as especificações do sistema também descrito em FIACRE. O programa em LD, traduzido para FIACRE, deve conter a lógica de controle e os processos representando a planta (com a descrição dos cilindros e ventosa), as especificações do sistema em FIACRE devem conter os processos com o comportamento dos cilindros e ventosa. Posteriormente é utilizado o compilador FIACRE/LOTOS para a entrada do CADP.

De forma similar aos processos da planta, referente ao programa de controle em LD, as especificações correspondentes às propriedades desejadas para o sistema SAP, escritas em FIACRE foram divididas em três processos: dois representando os cilindros (*3A1 e 3A2*) e um representando a ventosa (*3A3*). Também foi preciso criar um processo que simula a recepção e o envio dos sinais de entrada e saída do Módulo 3 (outros módulos do SAP completo, não estudado neste exemplo), de

acordo com a modelagem apresentada. Para criação dos processos, da mesma forma que os processos anteriores, foi seguida a descrição dos estados da modelagem, ou seja, cada estado do modelo corresponde a um estado do processo que representa o cilindro. A diferença entre as duas abordagens é que nas especificações das propriedades em FIACRE, o controle está embutido no comportamento dos cilindros, diferentemente do programa de controle do CLP, que está separado, atuando sobre a planta. O cilindro 2 (*3A2*) e a ventosa têm comportamento similar ao cilindro *3A1*.

O processo do ambiente para simulação (representação fictícia da comunicação que o módulo 3 deveria ter com os outros módulos do SAP) dos sinais de entrada e saída em FIACRE é o seguinte:

```
process ambiente [sinalMod3Ini, sinalMod3Fim: bool] (Mod3Ini, Mod3Fim:
bool) is

states inicio, fim
init to inicio

from inicio sinalMod3Ini !Mod3Ini;
  to fim
from fim sinalMod3Fim ?Mod3Fim;
  to inicio
```

O processo em FIACRE do comportamento do cilindro *3A1* das especificações das propriedades é o seguinte.

```
process A3_1 [sinalMod3Fim, sinal3_S2, sinal3_S3, sinalT2: bool]
(S3_3, S3_2, T2, Mod3Fim, A3_1, S3_1: bool) is

states cilindro1_recuado, cilindro1_avancado, avancando_cilindro1,
recuando_cilindro1
init to cilindro1_recuado

from cilindro1_recuado sinal3_S3 ?S3_3;
  if S3_3 then A3_1:= true; to avancando_cilindro1
  else to cilindro1_recuado end
from avancando_cilindro1 S3_2:= true; sinal3_S2 !S3_2;
  to cilindro1_avancado
```

```
from cilindro1_avancado sinalT2 ?T2;
  if T2 then A3_1:= false; to recuando_cilindro1
  else to cilindro1_avancado end
from recuando_cilindro1 Mod3Fim:= true; S3_1:= true; sinalMod3Fim !Mod3Fim;
  to cilindro1_recuado
```

Então com a tradução do programa de controle em LD, juntamente com a planta, para FIACRE e as especificações também em FIACRE, estes dois arquivos são traduzidos para arquivos na linguagem LOTOS. Na verificação por equivalências, é necessária a utilização da ferramenta CADP, em que é preciso fazer alguns passos anteriores, pois a equivalência é feita entre arquivos LOTOS e BCG. Para a transformação para BCG é preciso primeiro um arquivo *'.h'* através do comando *'caesar.adt'*, no CADP. Com esse arquivo, é possível então gerar o arquivo no formato BCG por intermédio do comando *'caesar -bcg'*. De posse destes arquivos, é possível através do *'bisimulator'* fazer a equivalência observacional entre as especificações descritas em LOTOS, apenas com as transições relevantes para o programa, e o programa no formato BCG. O resultado verdadeiro (*TRUE*) gerado pelo CADP, apresentado de forma integral no Apêndice B.3, comprova a equivalência entre os dois sistemas de acordo com a ferramenta de verificação e as propriedades especificadas.

5.4 Exemplo de verificação de um sistema misturador controlado por CLP

Nesta seção será apresentado outro exemplo ilustrativo de verificação formal de um sistema industrial controlado por CLP escrito em LD utilizando a cadeia de verificação por *model-checking*. Também serão apresentados os aspectos da transformação das propriedades e a aplicação destas no processo de verificação.

5.4.1 Descrição do sistema misturador

O exemplo desta seção foi apresentado por completo pela norma IEC 61131-3 [26] e tratado de forma resumida por Rossi e Schnoebelen [41]. O sistema apresentado é um subconjunto funcional do sistema *MIX2BRIX*, apresentado na norma, que controla a mistura de materiais sólidos para um sistema industrial de construção civil. O subconjunto apresentado é um misturador (*mixer*) de produtos que têm a origem do material em um sistema anterior e derrama para o próximo sistema da planta de produção. A partir do estado inicial (todos os atuadores parados e o misturador no sentido vertical), o botão de início é pressionado (*ST: 0->1*), iniciando o motor do misturador (*MR: 0->1*). Depois de um tempo *T1* decorrido, o compartimento começa a girar para o sentido horizontal (*MP1: 0->1*) até o sensor *S1* detectar que a posição foi atingida (*S1: 0->1*). Então a mistura é paralisada (*MR, MP1: 1->0*) e o misturador move-se de volta para sua posição original vertical (*MP0: 0->1*), até atingir o sensor, representado pelo sinal *S0*. A qualquer momento o sinal STOP pode substituir esse comportamento, e seu efeito é parar todos os atuadores, não importando o estado atual. Quando o sinal *STOP* desaparece, o sistema recupera seu estado inicial, a espera de uma nova ocorrência do sinal *ST*. A Fig. 5.9 ilustra fisicamente como é composto o misturador, e as entradas e saídas do sistema são apresentadas na Fig. 5.10.

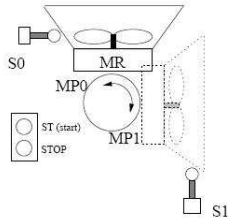


Figura 5.9: Modelo físico do misturador [41].

Para o exemplo do sistema para um misturador, podem ser descritas algumas propriedades analisando a descrição literal das especi-

Figura 5.10: Sistema de entradas e saídas do controlador CLP do misturador [41].

ficações do sistema. Uma propriedade importante, por exemplo, é que os sinais *MP0 e MP1* não ocorram ao mesmo tempo, podendo ocasionar defeito no motor bidirecional *MP*. Outra propriedade importante é a ordem de acionamento dos motores *MR e MP*, para que o sistema funcione corretamente.

O programa em LD para o controle do sistema misturador é apresentado na Fig. 5.11.

5.4.2 Modelagem dos motores do sistema misturador

Nesta subseção é apresentada a modelagem do ambiente externo referente ao exemplo do sistema para um misturador industrial. Este ambiente é representado por dois motores com os sinais *MP* (motor bidirecional) e *MR* (motor simples). O motor simples, representado por *MR*, possui os estados de *desligado e ligado*, e é acionado pelo recebimento do sinal *MR*, oriundo do programa de controle. A planta recebe o sinal, se *MR* estiver ativo (true), liga o motor, e permanece ligado. Se o sinal MR estiver desativado (false) desliga o motor, e permanece desligado. O motor bidirecional, acionado pelos sinais de *MP*, possui os estados *parado*, e os que significam o sentido de rotação, como *direita e esquerda*. O motor bidirecional, quando recebe o sinal *MP1*, se estiver ativo, gira para a direita até acionar o sensor *S1* (true) e parar o motor. Se o sinal *MP1* for desativado durante o percurso (*not MP1*) o motor desliga. Quando recebe o sinal *MP0*, se estiver ativo, gira para a esquerda até acionar o sensor *S0* (true) e parar o motor. Se o sinal *MP0* for desativado durante o percurso (*not MP0*) o motor desliga. As

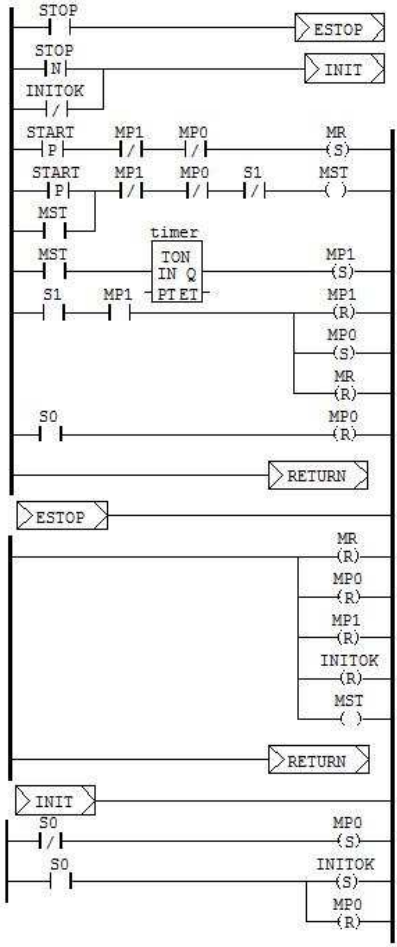


Figura 5.11: Programa em LD para o misturador [41].

Figuras 5.12 e 5.13 representam as modelagens dos motores *MR e MP*, respectivamente.

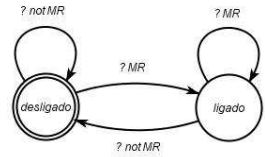


Figura 5.12: Comportamento do motor simples MR.

A partir do modelo do motor MP é possível também modelar os estados que não devem ser alcançados, e podem assim ocasionar uma falha. Por exemplo, partindo do estado girando para a direita, se receber um sinal MP0, e este estiver ativo, vai para o estado *quebrado*. O mesmo acontece partindo do estado girando para a esquerda, que se recebe um sinal MP1 ativado, também vai para estado *quebrado*. Este estado de falha é uma situação não desejada para o sistema, que deve ser garantida por meio do controle do CLP para nunca ocorrer.

Foram modelados também os botões de acionamento (*START*) e parada (*STOP*) do sistema. Se os sinais estão ativos (*START e STOP*), ou seja, com o valor *true*, ficam nos estados ativados e vão para os estados desativados quando os sinais forem desativados (*not ou false*), enviando os sinais com os seus valores para a leitura no ciclo de execução, simbolizando o início e fim do sistema por meio dos botões. As Figuras 5.14 e 5.15 representam as modelagens dos botões *START e STOP*, respectivamente.

A partir da modelagem, foi feita a especificação do comportamento dos motores através de processos em FIACRE. O processo em FIACRE que representa o comportamento do motor *MR* do misturador é o seguinte:

```
process motor_MR [sinal_MR: bool](MR: bool) is

states ligado, desligado
init to desligado
```

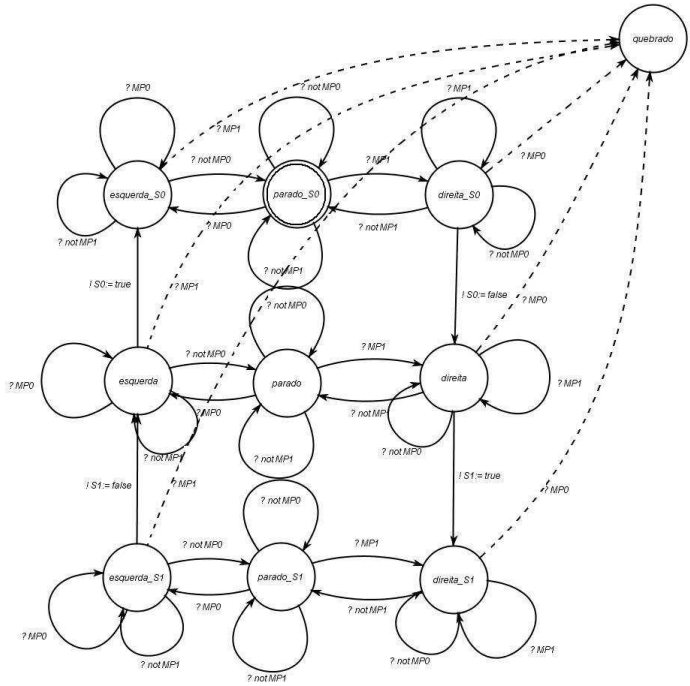


Figura 5.13: Comportamento do motor bidirecional MP.

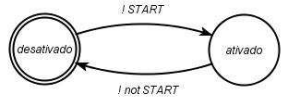


Figura 5.14: Comportamento do botão START.

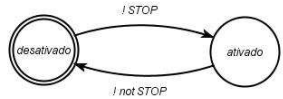


Figura 5.15: Comportamento do botão STOP.

```
from desligado sinal_MR ?MR;
  if MR then to ligado
  else to desligado end
from ligado sinal_MR ?MR;
  if MR then to ligado
  else to desligado end
```

O processo em FIACRE que representa o comportamento do motor *MP* do misturador é o seguinte:

```
process motor_MP [sinal_MP1, sinal_MP0, sinal_S1, sinal_S0: bool]
(MP0, MP1, S0, S1: bool) is

states parado, direita, esquerda, parado_S0, direita_S0, esquerda_S0,
parado_S1, direita_S1, esquerda_S1, quebrado
init to parado

from parado
  select
    sinal_MP0 ?MP0; if MP0 then to esquerda
                    else to parado end
    [] sinal_MP1 ?MP1; if MP1 then to direita
                       else to parado end
  end
from direita_S1
  select
    sinal_MP1 ?MP1; if MP1 then to direita_S1
                    else to parado_S1 end
    [] sinal_MP0 ?MP0; if MP0 then to quebrado
                       else to direita_S1 end
  end
from parado_S1
  select
    sinal_MP0 ?MP0; if MP0 then to esquerda_S1
                    else to parado_S1 end
    [] sinal_MP1 ?MP1; if MP1 then to direita_S1
                       else to parado_S1 end
  end
from esquerda_S1
  select
    sinal_MP0 ?MP0; if MP0 then to esquerda_S1
                    else to parado_S1 end
```

```
    [] sinal_S1 !S1:=false; to esquerda
    [] sinal_MP1 ?MP1; if MP1 then to quebrado
                       else to esquerda_S1 end
  end
from esquerda
  select
    sinal_MP0 ?MP0; if MP0 then to esquerda
                    else to parado end
    [] sinal_S0 !S0:=true; to esquerda_S0
    [] sinal_MP1 ?MP1; if MP1 then to quebrado
                       else to esquerda end
  end
from esquerda_S0
  select
    sinal_MP0 ?MP0; if MP0 then to esquerda_S0
                    else to parado_S0 end
    [] sinal_MP1 ?MP1; if MP1 then to quebrado
                       else to esquerda_S0 end
  end
from parado_S0
  select
    sinal_MP0 ?MP0; if MP0 then to esquerda_S0
                    else to parado_S0 end
    [] sinal_MP1 ?MP1; if MP1 then to direita_S0
                       else to parado_S0 end
  end
from direita_S0
  select
    sinal_MP1 ?MP1; if MP1 then to direita_S0
                    else to parado_S0 end
    [] sinal_S0 !S0:=false; to direita
    [] sinal_MP0 ?MP0; if MP0 then to quebrado
                       else to direita_S0 end
  end
from direita
  select
    sinal_MP1 ?MP1; if MP1 then to direita
                    else to parado end
    [] sinal_S1 !S1:=true; to direita_S1
    [] sinal_MP0 ?MP0; if MP0 then to quebrado
                       else to direita end
  end
```

Os processos em FIACRE que representam os comportamentos dos botões *START e STOP* são os seguintes:

```
process botao_START [sinal_start: bool](START: bool) is

states desativado, ativado
init to desativado

from desativado
  sinal_start !START:=true;
    to ativado
from ativado
  sinal_start !START:=false;
    to desativado


process botao_STOP [sinal_stop: bool](STOP: bool) is

states desativado, ativado
init to desativado

from desativado
  sinal_stop !STOP:=true;
    to ativado
from ativado
  sinal_stop !STOP:=false;
    to desativado
```

5.4.3 Modelagem e verificação das propriedades do sistema misturador

Para o exemplo do sistema para um misturador, por ser um controle menos complexo, algumas etapas não necessitam ser realizadas para a especificação das propriedades. Foram feitas as etapas de formulação da descrição do problema, com as especificações e requisitos do sistema, a elaboração da tabela de correspondência lógica e a modelagem dos motores que constituem o sistema. O Apêndice B.4 apresenta a tradução do sistema misturador completa em FIACRE, de acordo com as regras de transformação apresentadas no Capítulo 4, juntamente com

a planta.

De acordo com o que foi tratado na seção sobre propriedades específicas, para o exemplo do sistema misturador, podem ser observadas algumas propriedades de acordo com o objetivo do sistema. Uma propriedade importante é, de acordo com os sinais *MP0 e MP1*, não chegar no estado quebrado. Esta propriedade pode ser verificada tanto pelo estado do modelo da planta, quanto pelo estado das variáveis que representam os sinais MP0 e MP1.

O motor MP nunca pode estar girando para a esquerda e para a direita ao mesmo tempo, ou seja, no estado quebrado.

[] - (*motor_MP_quebrado*)

O motor *MP* girando para a esquerda implica em não chegar no estado quebrado.

[] (*motor_MP_esquerda* ⇒ - *motor_MP_quebrado*).

O motor *MP* girando para a direita implica em não chegar no estado quebrado.

[] (*motor_MP_direita* ⇒ - *motor_MP_quebrado*).

O resultado *TRUE* do verificador mostra que o controle do programa funciona corretamente.

Outra forma de verificação poderia ser feita através dos sinais, ou seja, os sinais que acionam o motor para direita e esquerda nunca estarão ativos ao mesmo tempo.

Conforme observado, outra propriedade importante é a ordem de acionamento dos motores *MR e MP*, para que o sistema funcione corretamente.

Após o acionamento do motor MP para a esquerda, então o motor MR acionará:

[] (*motor_MP_esquerda* ⇒ *motor_MR_ligado*)

Após o acionamento do motor MP para a direita, então o motor MR não acionará:

[] (*motor_MP_direita* ⇒ - *motor_MR_ligado*)

O resultado *TRUE* do verificador confirma o funcionamento correto de acordo com a descrição estrutural do sistema.

No Apêndice B.5 são apresentados os resultados completos das propriedades que foram verificadas na ferramenta SELT/TINA.

5.5 Conclusão do capítulo

Neste capítulo foi apresentada a verificação de propriedades aplicadas nos exemplos para CLPs, do sistema de automação pneumática (SAP) e do sistema para um misturador (*mixer*), ambos tendo o programa de controle (CLP) escrito em LD. Primeiramente, foram descritas as propriedades, gerais e específicas, dos sistemas controlados, os modelos das plantas e sua composição com os programas do CLP que implementam seu controle. Foi feita também a tradução destas propriedades de acordo com as abordagens por *model-checking* e equivalências (esta apenas para o SAP) e a verificação nos exemplos apresentados. Os resultados obtidos, de acordo com as ferramentas de verificação utilizadas, comprovaram que os programas estão corretos conforme as especificações desejadas.

Capítulo 6

Conclusão

A importância da padronização das linguagens de CLP, através da norma IEC 61131-3 é essencial para o desenvolvimento de programas visando um baixo custo e menor tempo de implementação, sem perder a qualidade. O estudo da arquitetura, modelo de software e linguagens de programação de CLPs facilitaram a modelagem dos programas, bem como a percepção das vantagens de utilização da norma, além disso, a ferramenta de programação *PLCOpen Editor*, pertencente ao ambiente *Beremiz*, é de fácil utilização para todos os tipos de programadores, além de ser software livre. A padronização utilizada pela ferramenta possibilitou a sua integração na cadeia de verificação de CLPs, com a utilização dos arquivos no formato XML representando os programas em LD. A opção da linguagem LD, como entrada das cadeias de verificação deste trabalho, deve-se ao fato de ser a mais antiga e amplamente utilizada pelos programadores de CLP.

A modelagem e verificação formal de programas para CLPs possibilitam que estes estejam de acordo com as especificações esperadas. O projeto *Topcased* proporciona este tipo de desenvolvimento para diversos sistemas, por meio de cadeias de verificação formal. Estas cadeias estão inseridas no contexto da engenharia dirigida a modelos, com o objetivo de transformar modelos próximos do usuário, em modelos

para aplicar a verificação formal. A inserção da programação de CLPs, pela linguagem LD, na entrada da transformação de modelos, possibilita uma nova opção de verificação de sistemas em um ambiente de desenvolvimento *Open Source* já consolidado. As cadeias de verificação propostas neste trabalho possibilitaram duas técnicas de verificação formal, bastante utilizadas para diversos tipos de sistemas: *model-checking* e equivalências de modelos. A linguagem intermediária FIACRE, inserida nestas cadeias, possibilita a integração de diversos tipos de modelos de entrada nas ferramentas de verificação.

A partir dos conceitos relacionados aos CLPs e suas formas de verificação e modelagem, juntamente com as definições da engenharia dirigida a modelos, foi possível caracterizar a transformação dos programas descritos na linguagem de entrada LD, para CLPs, para a linguagem intermediária FIACRE. Neste trabalho foram apresentadas as modelagens do comportamento de um programa em LD, que constituem a base para a transformação de modelos. Foram descritas as regras de tradução de programas em LD para FIACRE, de acordo com a modelagem proposta, juntamente com as construções em FIACRE dos principais elementos que compõem um programa de CLP. A partir do programa na linguagem LD, transformado para FIACRE, foi possível utilizar os compiladores já implementados (*FIACRE-TTS e FIACRE-LOTOS*) para utilização nas ferramentas de verificação formal (*TINA e CADP*).

A verificação de propriedades dos exemplos de programas para CLPs (sistema de automação pneumática e sistema para um misturador), descritos em LD, possibilitou as primeiras validações da modelagem de transformação e verificação propostas. Neste trabalho foram descritas as propriedades dos programas, gerais e específicas, e as modelagens dos elementos que compõem o ambiente externo dos exemplos. Foi feita também a tradução das propriedades de acordo com as abordagens por *model-checking* e equivalências, e as aplicações da verificação nos exemplos apresentados. Os resultados satisfatórios que foram obtidos, de acordo com as ferramentas de verificação, comprovaram que os programas estão conforme as suas especificações.

Com a conclusão deste trabalho, novas perspectivas e projeções para trabalhos futuros surgiram. A complementação deste trabalho, a partir dos resultados obtidos, pode ser feita através dos seguintes acréscimos e atividades futuras:

- Expansão das regras de transformação de mais elementos que constituem o LD, como por exemplo, acréscimo de blocos funcionais específicos;
- Aplicação de outros programas reais de CLPs na cadeia de verificação, que abrangem problemas diferentes de controle, e sobretudo, de maior porte, para poder analisar a escalabilidade do método e das ferramentas;
- Expansão da entrada da cadeia de verificação para todas as linguagens da norma IEC 61131-3;
- Finalização da implementação da transformação automática utilizando a linguagem ATL;
- Transformação automática das propriedades desejadas através da representação por matriz de causa e efeito.

Este trabalho também serve como base para que outras pesquisas e estudos possam ser realizados, tanto na modelagem de programas de CLP, como para o uso da verificação formal de sistemas. O propósito dos trabalhos futuros é deixar o trabalho mais abrangente e com um potencial maior de sua aplicação. Os resultados iniciais deste trabalho já geraram a publicação de artigo no Congresso Brasileiro de Automática 2010 [45].

Apêndice A

Anexos

As linguagens de programação de CLP da norma IEC 61131-3 foram apresentadas no decorrer do Capítulo 2 deste trabalho. Este anexo apresenta as principais instruções e operadores das linguagens para complementar o entendimento. Estas definições foram retiradas da norma IEC [26].

A.1 Operadores e instruções da IL

A IL é composta das seguintes estruturas de elementos de programa:

```
TYPE...END_TYPE
VAR...END_VAR
VAR_INPUT...END_VAR
VAR_OUTPUT...END_VAR
VAR_IN_OUT...END_VAR
VAR_EXTERNAL...END_VAR
VAR_TEMP...END_VAR
VAR_ACCESS...END_VAR
VAR_GLOBAL...END_VAR
VAR_CONFIG...END_VAR
FUNCTION ... END_FUNCTION
FUNCTION_BLOCK...END_FUNCTION_BLOCK
PROGRAM...END_PROGRAM
STEP...END_STEP
TRANSITION...END_TRANSITION
```

`ACTION...END_ACTION`

Os operadores encontrados na IL são:

- LD: Carrega o resultado corrente igual ao operando;
- ST: Armazena o resultado corrente no operando locado;
- S: Set o operando em 1 se o resultado corrente booleano é 1;
- R: Reset o operando em 0 se o resultado corrente booleano é 1;
- AND: Logica AND;
- OR: Logica OR;
- XOR: Logica OR exclusivo;
- NOT: Logica de negação (complemento de um);
- ADD: Adição;
- SUB: Subtração;
- MUL: Multiplicação;
- DIV: Divisão;
- MOD: Modulo-divisão;
- GT: Comparação >;
- GE: Comparação >=;
- EQ: Comparação =;
- NE: Comparação <>;
- LE: Comparação <=;
- LT: Comparação <;
- JMP: Salto para o label;

- CAL: chama o bloco de função.

Algumas das instruções encontradas no IL podem ser:

- expressões;
- operação de salto;
- chamada de bloco funcional;
- chamada de uma função formal;
- retorno de um operador.

A.2 Operadores e instruções do ST

Para a linguagem ST as estruturas de programação e os operadores são basicamente os mesmos da linguagem IL. As principais instruções do ST são as seguintes, juntamente com exemplos de aplicação:

- Atribuição:

```
A := B; CV := CV+1; C := SIN(X);
```

- Chamada de bloco de função e saída FB:

```
CMD_TMR(IN:=%IX5, PT:=T#300ms) ;
A := CMD_TMR.Q ;
```

- If:

```
D := B*B - 4*A*C ;
IF D < 0.0 THEN NROOTS := 0 ;
ELSIF D = 0.0 THEN
  NROOTS := 1 ;
  X1 := - B/(2.0*A) ;
ELSE
  NROOTS := 2 ;
  X1 := (- B + SQRT(D))/(2.0*A) ;
  X2 := (- B - SQRT(D))/(2.0*A) ;
END_IF ;
```

- Case:

```
TW := BCD_TO_INT(THUMBWHEEL);
TW_ERROR := 0;
CASE TW OF
  1,5: DISPLAY := OVEN_TEMP;
  2: DISPLAY := MOTOR_SPEED;
  3: DISPLAY := GROSS - TARE;
  4,6..10: DISPLAY := STATUS(TW - 4);
ELSE DISPLAY := 0 ;
  TW_ERROR := 1;
END_CASE;
QW100 := INT_TO_BCD(DISPLAY);
```

- For:

```
J := 101 ;
FOR I := 1 TO 100 BY 2 DO
  IF WORDS[I] = 'KEY' THEN
    J := I ;
    EXIT ;
  END_IF ;
END_FOR ;
```

- While:

```
J := 1;
WHILE J <= 100 & WORDS[J] <> 'KEY' DO
  J := J+2 ;
END_WHILE ;
```

- Repeat:

```
J := -1 ;
REPEAT
  J := J+2 ;
UNTIL J = 101 OR WORDS[J] = 'KEY'
END_REPEAT ;
```

A.3 Funções básicas para FBD e LD

Nesta seção do anexo são apresentados os principais blocos de funções utilizados em FBD e LD. Além desses, existem as funções iguais aos operadores descritos para as linguagens IL e ST. A Fig. A.1 apresenta os blocos funcionais biestáveis *set e reset*, ou seja, dois estados estáveis, em que, uma vez que o circuito for acionado permanecerá indefinidamente neste estado.

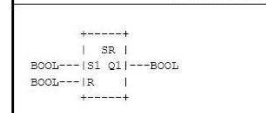
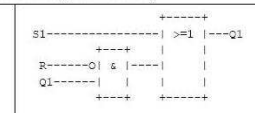
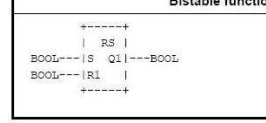
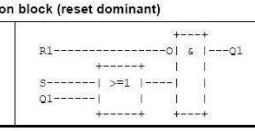
Graphical form	Function block body
Bistable function block (set dominant)	
	
Bistable function block (reset dominant)	
	

Figura A.1: Blocos de função set e reset.

A Fig. A.2 apresenta os blocos sensíveis a bordas de subida e descida.

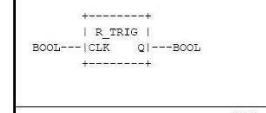
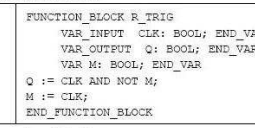
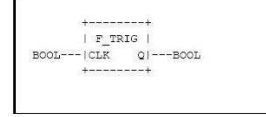
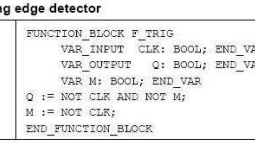
Graphical form	Definition ST language
Rising edge detector	
	
Falling edge detector	
	

Figura A.2: Blocos de função sensíveis a borda.

A Fig. A.3 apresenta o bloco para os temporizadores através de pulso, contagem crescente e decrescente.

A Fig. A.4 apresenta os blocos contadores com contagem crescente. A Fig. A.5 apresenta os blocos contadores com contagem decrescente. A Fig. A.6 apresenta os blocos contadores crescente e decrescente.

Graphical form	Description
+--------+ | *** | BOOL---|IN Q|---BOOL TIME---|PT ET|---TIME +--------+	*** is: TP (Pulse) TON (On-delay) T---0 (On-delay) TOF (Off-delay) 0---T (Off-delay)

Figura A.3: Blocos de função temporizadores.

Graphical form	Function block body ST language
Up-counter	
+-----+ | CTU | BOOL--->CU Q|---BOOL BOOL---|R | INT---|PV CV|---INT +-----+	IF R THEN CV := 0 ; ELSIF CU AND (CV < PVmax) THEN CV := CV+1; END_IF ; Q := (CV >= PV) ;
+----------+ | CTU_DINT | BOOL--->CU Q|---BOOL BOOL---|R | DINT---|PV CV|---DINT +----------+	Same as 1a
+----------+ | CTU_LINT | BOOL--->CU Q|---BOOL BOOL---|R | LINT---|PV CV|---LINT +----------+	Same as 1a
+-----------+ | CTU_UDINT | BOOL--->CU Q|---BOOL BOOL---|R | UDINT---|PV CV|---UDINT +-----------+	Same as 1a
+-----------+ | CTU_ULINT | BOOL--->CU Q|---BOOL BOOL---|R | ULINT---|PV CV|---ULINT +-----------+	Same as 1a

Figura A.4: Blocos de função contadores up.

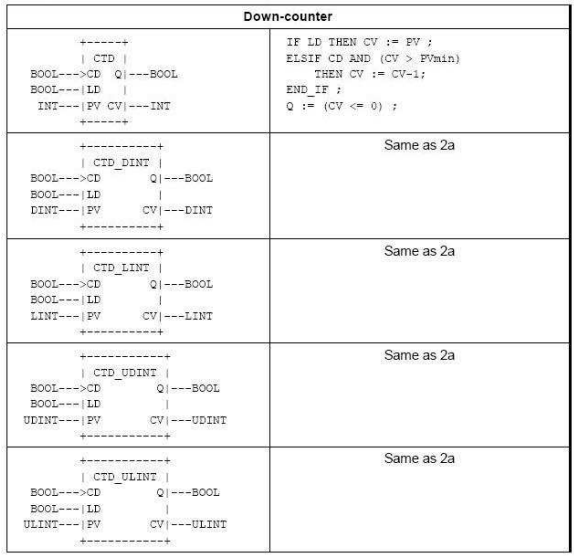


Figura A.5: Blocos de função contadores down.

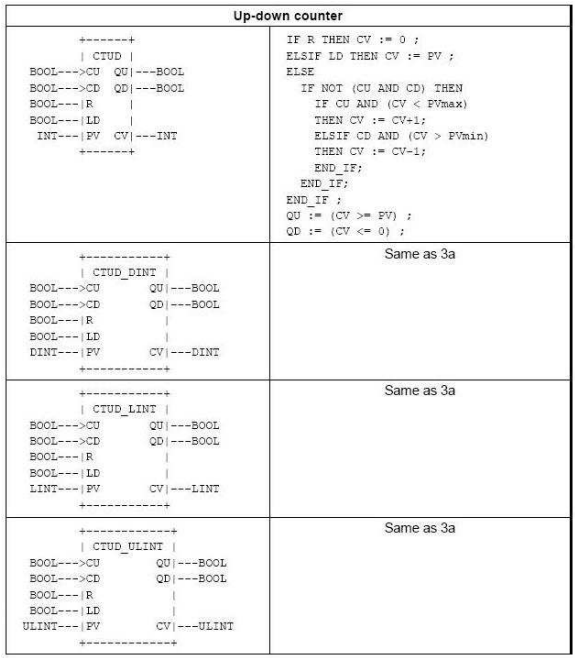


Figura A.6: Blocos de função contadores up-down.

Apêndice B

Apêndices

Este apêndice descreve os códigos em FIACRE dos programas de CLP apresentados nos exemplos do Capítulo 5 e os resultados completos gerados pelas ferramentas de verificação.

B.1 Programa em FIACRE do SAP

Nesta seção do apêndice é apresentado o programa em FIACRE do exemplo do SAP, juntamente com a sua planta. Este programa é resultado da transformação de LD para FIACRE e possui o seguinte código:

```
/* ------------------------------------------------------------------------- */
/*                 Programa equivalente ao LD do Módulo 3                    */
/* ------------------------------------------------------------------------- */
/* ------------------------------------------------------------------------- */
/*          Processo que descreve o funcionamento principal do LD            */
/* ------------------------------------------------------------------------- */

process ciclo_execucao [rung_delay, novo_ciclo: none, leitura_3S4, leitura_3S3,
leitura_T1, leitura_T2, leitura_3S2, leitura_Mod3Fim, escrita_T1, escrita_T2,
escrita_3S2, escrita_3S3, escrita_3S4, escrita_Mod3Ini: bool]
(Mod3Ini, Mod3Fim, Manual1, Manual2, Manual3, B21, B22, T1, T2, S0_1, S3_1, S3_2,
S3_3, S3_4, A3_1, A3_2, A3_3: bool) is

/* Estados do sistema */
```

```
states rung1, rung2, rung3, rung4, rung5, rung6, rung7, rung8, rung9, rung10,
rung11, rung12, rung13, rung14, rung15, rung16, rung17, inicio, leitura1, leitura2,
leitura3, leitura4, leitura5, fim1, fim2, escrita1, escrita2, escrita3, escrita4,
escrita5, escrita6

/* Variáveis auxiliares */

var
        P0: bool:= false,
        P1: bool:= false,
        P2: bool:= false,
        P3: bool:= false,
        P4: bool:= false,
        P5: bool:= false,
        P6: bool:= false,
        P7: bool:= false

/* Estado inicial */

init to inicio

/* Leitura dos dados de entrada */

from inicio
   leitura_3S4 ?S3_4; to leitura1
from leitura1
  leitura_3S3 ?S3_3; to leitura2
from leitura2
  leitura_T1 ?T1; to leitura3
from leitura3
  leitura_T2 ?T2; to leitura4
from leitura4
  leitura_3S2 ?S3_2; to leitura5
from leitura5
  leitura_Mod3Fim ?Mod3Fim; to rung1

/* Início da execução do programa */

from rung1 rung_delay;
        B21:= Manual2 and not B22; to rung2
from rung2 rung_delay;
        B22:= Manual2 or B21; to rung3
from rung3 rung_delay;
        P0:= ((not P6 and P7 and ((Manual1 and Manual3) or B21)) or S0_1 or P0) and
        not P1; to rung4
from rung4 rung_delay;
        P1:= ((not P7 and P0 and ((Manual1 and Manual3 and Mod3Ini) or B21)) or P1)
        and not P2 and not S0_1; to rung5
from rung5 rung_delay;
```

```
        P2:= ((not P0 and P1 and ((Manual1 and Manual3 and S3_4) or B21)) or P2) and
        not P3 and not S0_1; to rung6
from rung6 rung_delay;
        P3:= ((not P1 and P2 and ((Manual1 and Manual3 and T1) or B21)) or P3) and
        not P4 and not S0_1; to rung7
from rung7 rung_delay;
        P4:= ((not P2 and P3 and ((Manual1 and Manual3 and S3_3) or B21)) or P4) and
        not P5 and not S0_1; to rung8
from rung8 rung_delay;
        P5:= ((not P3 and P4 and ((Manual1 and Manual3 and S3_2) or B21)) or P5) and
        not P6 and not S0_1; to rung9
from rung9 rung_delay;
        P6:= ((not P4 and P5 and ((Manual1 and Manual3 and T2) or B21)) or P6) and
        not P7 and not S0_1; to rung10
from rung10 rung_delay;
        P7:= ((not P5 and P6 and ((Manual1 and Manual3 and S3_1) or B21)) or P7) and
        not P0 and not S0_1; to rung11
from rung11
        rung_delay; A3_2:= P1 or A3_2; to rung12
from rung12
        rung_delay; A3_2:= not P3 and A3_2; to rung13
from rung13
        rung_delay; A3_3:= P2 or A3_3; to rung14
from rung14
        rung_delay; A3_3:= not P5 and A3_3; to rung15
from rung15
        rung_delay; A3_1:= P4 or A3_1; to rung16
from rung16
        rung_delay; A3_1:= not P6 and A3_1; to rung17
from rung17 rung_delay;
        Mod3Fim:= P7; to fim1
from fim1 rung_delay;
        S0_1:= false; to escrita1

/* Escrita dos dados de saída */

from escrita1
   escrita_Mod3Ini !Mod3Ini; Mod3Ini:= false; to escrita2
from escrita2
   escrita_T1 !T1; to escrita3
from escrita3
   escrita_3S4 !S3_4; to escrita4
from escrita4
   escrita_3S2 !S3_2; to escrita5
from escrita5
   escrita_3S3 !S3_3; to escrita6
from escrita6
   escrita_T2 !T2; to fim2
```

```
/* Reinício do ciclo */

from fim2  novo_ciclo;
        to inicio

/* ------------------------------------------------------------------------- */
/*         Processo que descreve glue dos sinais de entrada                  */
/* ------------------------------------------------------------------------- */

process inputsGlue [leitura_3S4, leitura_3S3, leitura_T1, leitura_T2, leitura_3S2,
leitura_Mod3Fim, sinal3S4, sinal3S3, sinalT1, sinalT2, sinal3S2, sinalMod3Fim: bool]
(S3_4, S3_3, T1, T2, S3_2, Mod3Fim:bool) is

states inicio

init to inicio

/* Recebimento dos sinais oriundos da planta e envio para o ciclo de execução */

from inicio
        select
                leitura_3S4 !S3_4; to inicio
                [] leitura_3S3 !S3_3; to inicio
                [] leitura_T1 !T1; to inicio
                [] leitura_T2 !T2; to inicio
                [] leitura_3S2 !S3_2; to inicio
                [] leitura_Mod3Fim !Mod3Fim; to inicio
                [] sinal3S4 ?S3_4; to inicio
                [] sinal3S3 ?S3_3; to inicio
                [] sinalT1 ?T1; to inicio
                [] sinalT2 ?T2; to inicio
                [] sinal3S2 ?S3_2; to inicio
                [] sinalMod3Fim ?Mod3Fim; to inicio
        end

/* ------------------------------------------------------------------------- */
/*         Processo que descreve glue dos sinais de saída                    */
/* ------------------------------------------------------------------------- */

process outputsGlue [escrita_Mod3Ini, escrita_T1, escrita_3S4, escrita_3S2,
escrita_3S3, escrita_T2, sinalMod3Ini, sinalT1, sinal3S4, sinal3S2, sinal3S3,
sinalT2: bool] (Mod3Ini, T1, S3_4, S3_2, S3_3, T2:bool) is

states inicio

init to inicio

/* Recebimento dos sinais oriundos do ciclo de execução e envio para a  planta */
```

```
from inicio
        select
                escrita_Mod3Ini ?Mod3Ini; to inicio
                [] escrita_T1 ?T1; to inicio
          [] escrita_3S4 ?S3_4; to inicio
                [] escrita_3S2 ?S3_2; to inicio
                [] escrita_3S3 ?S3_3; to inicio
                [] escrita_T2 ?T2; to inicio
                [] sinalMod3Ini !Mod3Ini; to inicio
                [] sinalT1 !T1; to inicio
                [] sinal3S4 !S3_4; to inicio
                [] sinal3S2 !S3_2; to inicio
                [] sinal3S3 !S3_3; to inicio
                [] sinalT2 !T2; to inicio
        end

/* ------------------------------------------------------------------------- */
/*        Processo que representa a planta do cilindro 3A1                   */
/* ------------------------------------------------------------------------- */

process planta_cilindro1 [sinal3S3, sinal3S2, sinalT2, sinalMod3Fim: bool]
(S3_3, S3_2, T2, Mod3Fim: bool) is

states cilindro1_recuado, cilindro1_avancado, avancando_cilindro1,
recuando_cilindro1

init to cilindro1_recuado

from cilindro1_recuado sinal3S3 ?S3_3;
        to avancando_cilindro1
from avancando_cilindro1 S3_2:= true; sinal3S2 !S3_2;
        to cilindro1_avancado
from cilindro1_avancado sinalT2 ?T2;
        to recuando_cilindro1
from recuando_cilindro1 Mod3Fim:= true; sinalMod3Fim !Mod3Fim;
        to cilindro1_recuado

/* ------------------------------------------------------------------------- */
/*        Processo que representa a planta do cilindro 3A2                   */
/* ------------------------------------------------------------------------- */

process planta_cilindro2 [sinalMod3Ini, sinal3S4, sinalT1, sinal3S3: bool]
(Mod3Ini, S3_4, T1, S3_3: bool) is

states cilindro2_recuado, cilindro2_avancado, avancando_cilindro2,
recuando_cilindro2

init to cilindro2_recuado
```

```
from cilindro2_recuado sinalMod3Ini ?Mod3Ini;
        to avancando_cilindro2
from avancando_cilindro2 S3_4:= true; sinal3S4 !S3_4;
        to cilindro2_avancado
from cilindro2_avancado sinalT1 ?T1;
        to recuando_cilindro2
from recuando_cilindro2 S3_3:= true; sinal3S3 !S3_3;
        to cilindro2_recuado

/* ------------------------------------------------------------------------ */
/*       Processo que representa a planta da ventosa 3A3                    */
/* ------------------------------------------------------------------------ */

process planta_ventosa [sinal3S4, sinalT1, sinal3S2, sinalT2: bool, timerT1,
timerT2: none] (S3_4, T1, S3_2, T2: bool) is

states ventosa_desligada, ventosa_acionada, acionando_ventosa, desligando_ventosa,
leitura_inicio, leitura_fim

init to ventosa_desligada

from ventosa_desligada sinal3S4 ?S3_4;
        to acionando_ventosa
from acionando_ventosa timerT1; T1:= true;
        to leitura_inicio
from leitura_inicio sinalT1 !T1;
        to ventosa_acionada
from ventosa_acionada sinal3S2 ?S3_2;
        to desligando_ventosa
from desligando_ventosa timerT2; T2:= true;
        to leitura_fim
from leitura_fim sinalT2 !T2;
        to ventosa_desligada

/* ------------------------------------------------------------------------ */
/*  Componente que representa o main do sistema e a composição dos processos  */
/* ------------------------------------------------------------------------ */

component main is

/* Declaração dos intervalos de tempo de cada transição */

port

        novo_ciclo:      none in [1,1],
        rung_delay:      none in [0,0],
        sinalT1:         bool in [0,0],
        sinalT2:         bool in [0,0],
        sinal3S2:        bool in [0,0],
```

```
        sinal3S3:        bool in [0,0],
        sinal3S4:        bool in [0,0],
        sinalMod3Ini:    bool in [0,0],
        sinalMod3Fim:    bool in [0,0],
        leitura_3S2:     bool in [0,0],
        leitura_Mod3Fim: bool in [0,0],
        leitura_3S4:     bool in [0,0],
        leitura_3S3:     bool in [0,0],
        leitura_T1:      bool in [0,0],
        leitura_T2:      bool in [0,0],
        escrita_3S2:     bool in [0,0],
        escrita_Mod3Ini: bool in [0,0],
        escrita_3S4:     bool in [0,0],
        escrita_3S3:     bool in [0,0],
        escrita_T1:      bool in [0,0],
        escrita_T2:      bool in [0,0],
        timerT1:         none in [2,2],
        timerT2:         none in [2,2]

/* Declaração dos processos do sistema para ocorrer a comunicação através do
operador de comunicação paralela par */

par

        rung_delay, novo_ciclo, leitura_3S4, leitura_3S3, leitura_T1,leitura_T2,
        leitura_3S2, leitura_Mod3Fim, escrita_T1, escrita_T2, escrita_3S2,
        escrita_3S3, escrita_3S4, escrita_Mod3Ini
        -> ciclo_execucao [rung_delay, novo_ciclo, leitura_3S4, leitura_3S3,
        leitura_T1, leitura_T2,         leitura_3S2, leitura_Mod3Fim, escrita_T1,
        escrita_T2, escrita_3S2, escrita_3S3, escrita_3S4, escrita_Mod3Ini]
        (true, false, true, false, true, false, false, false, false, true, false,
        false, false, false, false, false, false)
          || sinal3S3, sinal3S2, sinalT2, sinalMod3Fim -> par
          -> planta_cilindro1 [sinal3S3, sinal3S2, sinalT2, sinalMod3Fim]
          (false, false, false, false)
          || sinalMod3Ini, sinal3S4, sinalT1, sinal3S3
          -> planta_cilindro2 [sinalMod3Ini, sinal3S4, sinalT1, sinal3S3]
          (false, false, false, false)
          || sinal3S4, sinalT1, sinal3S2, sinalT2, timerT1, timerT2
          -> planta_ventosa [sinal3S4, sinalT1, sinal3S2, sinalT2, timerT1, timerT2]
          (false, false, false, false)
        end
        || leitura_3S4, leitura_3S3, leitura_T1, leitura_T2, leitura_3S2,
        leitura_Mod3Fim, sinal3S4, sinal3S3, sinalT1, sinalT2, sinal3S2, sinalMod3Fim
        -> inputsGlue [leitura_3S4, leitura_3S3, leitura_T1, leitura_T2, leitura_3S2,
        leitura_Mod3Fim, sinal3S4, sinal3S3, sinalT1, sinalT2, sinal3S2, sinalMod3Fim]
        (false, false, false, false, false, false)
        || escrita_Mod3Ini, escrita_T1, escrita_3S4, escrita_3S2, escrita_3S3,
        escrita_T2, sinalMod3Ini, sinalT1, sinal3S4, sinal3S2, sinal3S3, sinalT2
```

```
        -> outputsGlue [escrita_Mod3Ini, escrita_T1, escrita_3S4, escrita_3S2,
        escrita_3S3, escrita_T2, sinalMod3Ini, sinalT1, sinal3S4, sinal3S2, sinal3S3,
        sinalT2] (false, false, false, false, false, false)
end
main
```

B.2 Resultados da ferramenta TINA para a verificação do SAP

Nesta seção são apresentados os resultados completos da verificação por *model-checking* aplicado no exemplo do SAP utilizando a ferramenta SELT/TINA. No resultado gerado pela ferramenta tem o nome da versão utilizada, o número de estados e transições do arquivo verificado, a resposta da verificação (*TRUE ou FALSE*) e o tempo para fazer a verificação. Se a resposta for *FALSE*, gera um contra-exemplo com o caminho onde a propriedade não é verificada, com o número do estado e a situação dos estados e variáveis no momento onde a propriedade não é verificada. Para este exemplo, as propriedades verificadas como falsas, garantem também o funcionamento correto. É possível ainda verificar se os elementos da planta estão atuando corretamente, através do relatório gerado pelo TINA de análise de alcançabilidade (levando em consideração também os dados) seguindo as classes (estado, condições das variáveis e transição) percorridas durante a execução.

Primeiramente é aplicada a fórmula para verificar se o ciclo de execução é sempre completo:

```
[] (main_inicio => <> main_fim)
Selt version 2.9.6 -- 06/10/09 -- LAAS/CNRS
ktz loaded, 218 states, 218 transitions
0.000s
TRUE
0.016s.
```

Na seqüência, a verificação da ordem dos acionamentos. Após o avanço do cilindro 2, então a ventosa acionará:

```
[] (avancando_cilindro2 => <> acionando_ventosa)
```

```
TRUE
0.031s.
```

Após o acionamento da ventosa, então o cilindro 1 avançará:

```
[] (acionando_ventosa => <> avancando_cilindro1)

TRUE
0.016s.
```

A propriedade anterior não é valida para, após o acionamento da ventosa, então o cilindro 2 avançará, e gera o seguinte contra-exemplo com o caminho onde a propriedade não é verificada:

```
[] (acionando_ventosa => <> avancando_cilindro2)

FALSE
state 0: inicio cilindro1_recuado cilindro2_recuado ventosa_desligada
main_vMod3Ini main_vS0_1  -main_1_t0 ... (preserving T)->
state 76: leitura1 cilindro1_recuado cilindro2_avancado acionando_ventosa
main_vS3_4 main_vA3_2 main_vA3_3 main_vP1 main_vP2 main_vaux_ventosa
-planta_ventosa_t1_main_t4 ...
(preserving - avancando_cilindro2 /\ acionando_ventosa)->
state 218: leitura2 cilindro1_recuado cilindro2_avancado ventosa_acionada
main_vT1 main_vS3_4 main_vA3_2 main_vA3_3 main_vP1 main_vP2 main_vaux_ventosa
-main_t6 ... (preserving - avancando_cilindro2)-> * [accepting]
state 334: rung16 cilindro1_recuado cilindro2_recuado ventosa_desligada
main_vT1 main_vT2 main_vS3_2 main_vS3_3 main_vS3_4 main_vP6 -main_t24 ...
(preserving - avancando_cilindro2)->
state 334: rung16 cilindro1_recuado cilindro2_recuado ventosa_desligada
main_vT1 main_vT2 main_vS3_2 main_vS3_3 main_vS3_4 main_vP6
0.016s.
```

A propriedade não é valida para, após o avanço do cilindro 1, então a ventosa acionará, e gera o seguinte contra-exemplo com o caminho onde a propriedade não é verificada:

```
[] (avancando_cilindro1 => <> acionando_ventosa)

FALSE
state 0: inicio cilindro1_recuado cilindro2_recuado ventosa_desligada
main_vMod3Ini main_vS0_1 -main_1_t0 ... (preserving T)->
state 127: leitura2 avancando_cilindro1 cilindro2_recuado ventosa_acionada
main_vT1 main_vS3_3 main_vS3_4 main_vA3_1 main_vA3_3 main_vP3
main_vP4main_vaux_cilindro1 -planta_cilindro1_t1_main_t7 ...
(preserving - avancando_cilindro3 /\ avancando_cilindro1)->
state 218: rung1 cilindro1_avancado cilindro2_recuado ventosa_acionada
main_vT1 main_vS3_2 main_vS3_3 main_vS3_4 main_vA3_1 main_vA3_3 main_vP3
```

```
main_vP4 main_vaux_cilindro1  -main_t9 ...
(preserving - acionando_ventosa)->* [accepting]
state 283: rung16 cilindro1_recuado cilindro2_recuado ventosa_deligada
main_vT1 main_vT2 main_vS3_2 main_vS3_3 main_vS3_4 main_vP6
-main_t24 ... (preserving -acionando_ventosa)->
state 283: rung16 cilindro1_recuado cilindro2_recuado ventosa_deslgada
main_vT1 main_vT2 main_vS3_2 main_vS3_3 main_vS3_4 main_vP6
0.000s.
```

Outra propriedade é se, em alguma situação, pode ocorrer colisão dos cilindros, ou seja, estarem acionados ao mesmo tempo. Por exemplo, o cilindro 1 e o cilindro 2 nunca estarão avançados ao mesmo tempo:

```
[] - (cilindro2_avancado /\ cilindro1_avancado)

TRUE
0.000s.
```

A propriedade anterior não é garantida para o cilindro 2 e a ventosa, e gera o seguinte contra-exemplo com o caminho onde a propriedade não é verificada:

```
[] - (cilindro2_avancado /\ vetosa_acionada)

FALSE
state 0:inicio cilindro1_recuado cilindro2_recuado cilindro3_recuado
main_vMod3Ini main_vS0__1  -main_t0 ... (preserving T)->
state 96: escrita1 cilindro1_recuado cilindro2_avancado ventosa_acionada
main_vT1 main_vS3_4 main_vA3_3 main_vP2 main_vP3 main_vaux_cilindro2*2
-planta_cilindro2_t2_main_t28
... (preserving cilindro2_avancado /\ ventosa_acionada)->
state 218: escrita2 cilindro1_recuado recuando_cilindro2 ventosa_acionada
main_vT1 main_vS3_4 main_vA3_3 main_vP2 main_vP3 main_vaux_cilindro2*2
[accepting all]
0.016s.
```

A propriedade anterior não é garantida para o cilindros 1 e a ventosa, e gera o seguinte contra-exemplo com o caminho onde a propriedade não é verificada:

```
[] - (cilindro1_avancado /\ ventosa_acionada)

FALSE
state 0: inicio cilindro1_recuado cilindro2_recuado ventosa_desligada
main_vMod3Ini main_vS0_1  -main_1_t0 ... (preserving T)->
state 147: escrita2 cilindro1_avancado cilindro2_recuado ventosa_acionada
main_vT1 main_vS3_2 main_vS3_3 main_vS3_4 main_vA3_1 main_vP4 main_vP5
```

```
main_vaux_cilindro3*2  -planta_cilindro3_t2_main_t31
... (preserving ventosa_acionada /\ cilindro1_avancado)->
state 218: escrita3 cilindro1_avancado cilindro2_recuado desligando_ventosa
main_vT1 main_vS3_2 main_vS3_3 main_vS3_4 main_vA3_1 main_vP4 main_vP5
main_vaux_cilindro3*2  [accepting all]
0.016s.
```

Os dois cilindros e a ventosa nunca estarão ativos ao mesmo tempo:

```
[] - (cilindro1_avancado /\cilindro2_avancado /\ ventosa_acionada)

TRUE
0.000s.
```

B.3 Resultado da ferramenta CADP para a verificação do SAP

Nesta seção é apresentado o resultado completo da verificação por equivalência de modelos aplicada no exemplo do SAP utilizando a ferramenta CADP. O comando executado no CADP gera o seguinte resultado, com o programa e a sua especificação:

```
caesar.open especificacao.lotos bisimulator -observational programa.bcg
caesar.open: using ‘‘/usr/cadp/bin.iX86/bisimulator.a’’
-- caesar.adt 5.3 -- H. Garavel, R. Mateescu, M. Sighireanu & Ph. Turlier --
caesar.adt : analyse syntaxique de ‘‘especificacao’’
caesar.adt : (‘‘especificacao.h’’ existe deja et est a jour)
-- caesar 7.1 -- Hubert Garavel (INRIA Rhone-Alpes) --
caesar : analyse syntaxique de ‘‘especificacao’’
caesar : (‘‘especificacao.c’’ existe deja et est a jour)
caesar.open: using link mode
caesar.open: (‘‘especificacao.o’’ already exists and is up to date)
/usr/cadp/src/com/cadp_cc  especificacao.o /usr/cadp/bin.iX86/bisimulator.a
-o bisimulator -L/usr/cadp/bin.iX86 -lcaesar -L/usr/cadp/bin.iX86
-lBCG_IO -lBCG -lm caesar.open:
running ‘‘bisimulator -observational programa.bcg’’ for ‘‘especificacao.lotos’’
TRUE
```

B.4 Programa em FIACRE do sistema misturador

Nesta seção é apresentado o programa em FIACRE do exemplo do misturador, juntamente com a sua planta. Este programa é resultado da transformação de LD para FIACRE e possui o seguinte código:

```
/* ------------------------------------------------------------------------- */
/*       Processo que descreve o ciclo de execução do programa LD            */
/* ------------------------------------------------------------------------- */

process ciclo [restart, rung_delay, final :none, timer1, timer2, escrita_MR,
escrita_MP0, escrita_MP1, escrita_S1, escrita_S0, leitura_MP1, leitura_MP0,
leitura_MR, leitura_S1, leitura_S0, leitura_start, leitura_stop :bool]
(MR, STOP, MP0, MP1, INITOK, MST, S0, START, S1, IN, Q: bool) is

states idle, rung1, rung2, rung3, rung4, rung5, rung5_2, rung5_3, rung5_4, rung6,
rung7, leitura1, leitura2, leitura3, leitura4, leitura5, leitura6, escrita1,
escrita2, escrita3, escrita4, escrita5, fim, RETURN, ESTOP, INIT, INIT2

        var
          pre_stop: bool:= false,
          pre_start: bool:= false

        init to idle

  from idle
    leitura_start ?START to leitura1
  from leitura1
    leitura_stop ?STOP to leitura2
  from leitura2
    leitura_S1 ?S1; to leitura3
  from leitura3
    leitura_S0 ?S0; to leitura4
  from leitura4
    leitura_MR ?MR; to leitura5
  from leitura5
    leitura_MP0 ?MP0; to leitura6
  from leitura6
    leitura_MP1 ?MP1; to rung1
  from rung1 rung_delay;
     if STOP then to ESTOP
     else to rung2 end
  from rung2 rung_delay;
     if (not STOP and pre_stop) or not INITOK then
         to INIT
     else to rung3 end
```

```
from rung3 rung_delay;
   MR:= ((START and not pre_start) and not MP1 and not MP0) or MR;
     to rung4
from rung4 rung_delay;
   MST:= ((START and not pre_start) or MST) and not MP1 and not MP0
   and not S1;
     to rung5
from rung5 rung_delay;
   IN:= MST;
     to rung5_2
from rung5_2
   timer1 !IN;
     to rung5_3
from rung5_3
   timer2 ?Q;
     to rung5_4
from rung5_4 rung_delay
   MP1:= Q or MP1;
     to rung6
from rung6 rung_delay;
   MP0:= (S1 and MP1) or MP0;
   MR:= not (S1 and MP1) and MR;
   MP1:= not (S1 and MP1) and MP1;
     to rung7
from rung7 rung_delay;
   MP0:= not S0 and MP0;
     to RETURN
from ESTOP
   MR:= false;
   MP0:= false;
   MP1:= false;
   INITOK:= false;
   MST:= true;
     to rung2
from INIT
   MP0:= not S0 or MP0;
     to INIT2
from INIT2
   INITOK:= S0 or INITOK;
   MP0:= not S0 and MP0;
     to RETURN
from RETURN
  escrita_S1 !S1; to escrita1
from escrita1
  escrita_S0 !S0; to escrita2
from escrita2
  escrita_MR !MR; to escrita3
from escrita3
  escrita_MP0 !MP0; to escrita4
```

```
  from escrita4
    escrita_MP1 !MP1; to escrita5
  from escrita5 rung_delay;
     pre_stop:= STOP;
     pre_start:= START;
       to fim
  from fim restart; to idle

/* ---------------------------------------------------------------------------- */
/*       Processo que descreve o funcionamento do temporizador                  */
/* ---------------------------------------------------------------------------- */

process ton [timer1, timer2: bool, timeout: none](IN, Q: bool)is

  states idle, running, timeout

  init to idle

  from idle
    select
      timer1 ?IN; if IN then to running
                  else loop end
      [] timer2 !Q; loop
    end
  from running
    select
      timer1 ?IN; if not IN then Q:= false; to idle
                  else loop end
      [] timeout; Q:= true; to timeout
      [] timer2 !Q; loop
    end
  from timeout
    select
      timer1 ?IN; if not IN then Q:= false; to idle
                  else loop end
      [] timer2 !Q; loop
    end

/* ---------------------------------------------------------------------------- */
/*       Processo que descreve glue dos sinais de entrada                       */
/* ---------------------------------------------------------------------------- */

process inputsGlue [leitura_S1, leitura_S0, leitura_MR, leitura_MP0, leitura_MP1,
leitura_start, leitura_stop, sinal_S1, sinal_S0, sinal_MR, sinal_MP0, sinal_MP1,
sinal_start, sinal_stop: bool] (S0, S1, MR, MP0, MP1, START, STOP:bool) is

        states inicio

        init to inicio
```

```
/* Recebimento dos sinais oriundos da planta e envio para o ciclo de execução*/

        from inicio
                select
                        leitura_S1 !S1; to inicio
                        [] leitura_S0 !S0; to inicio
                        [] leitura_MR !S0; to inicio
                        [] leitura_MP0 !S0; to inicio
                        [] leitura_MP1 !S0; to inicio
                        [] leitura_start !START to inicio
                        [] leitura_stop !STOP to inicio
                        [] sinal_S1 ?S1; to inicio
                        [] sinal_S0 ?S0; to inicio
                        [] sinal_MR ?MR; to inicio
                        [] sinal_MP0 ?MP0; to inicio
                        [] sinal_MP1 ?MP1; to inicio
                        [] sinal_start ?START; to inicio
                        [] sinal_stop ?STOP; to inicio
                end

/* ------------------------------------------------------------------------- */
/*       Processo que descreve glue dos sinais de saída                      */
/* ------------------------------------------------------------------------- */

process outputsGlue [escrita_MR, escrita_S0, escrita_S1, escrita_MP1,
escrita_MP0, sinal_MR, sinal_S0, sinal_S1, sinal_MP1, sinal_MP0: bool]
(MR, MP0, MP1, S0, S1:bool) is

        states inicio

        init to inicio

/* Recebimento dos sinais oriundos do ciclo de execução e envio para a planta */

        from inicio
                select
                        escrita_MR ?MR; to inicio
                        [] escrita_S0 ?S0; to inicio
                        [] escrita_S1 ?S1; to inicio
                        [] escrita_MP1 ?MP1; to inicio
                        [] escrita_MP0 ?MP0; to inicio
                        [] sinal_MR !MR; to inicio
                        [] sinal_S0 !S0; to inicio
                        [] sinal_S1 !S1; to inicio
                        [] sinal_MP1 !MP1; to inicio
                        [] sinal_MP0 !MP0; to inicio
                end
```

```
/* ------------------------------------------------------------------------ */
/*       Processo que representa a planta motor do misturador               */
/* ------------------------------------------------------------------------ */

process motor_MR [sinal_MR: bool](MR: bool) is

  states ligado, desligado
  init to desligado

  from desligado sinal_MR ?MR;
    if MR then to ligado
    else to desligado end
  from ligado sinal_MR ?MR;
    if MR then to ligado
    else to desligado end

/* ------------------------------------------------------------------------ */
/*      Processo que representa a planta motor bidirecional                 */
/* ------------------------------------------------------------------------ */

process motor_MP [sinal_MP1, sinal_MP0, sinal_S1, sinal_S0: bool]
(MP0, MP1, S0, S1: bool) is

  states parado, direita, esquerda, parado_S0, direita_S0, esquerda_S0,
  parado_S1, direita_S1, esquerda_S1, quebrado
  init to parado

  from parado
    select
      sinal_MP0 ?MP0; if MP0 then to esquerda
                      else to parado end
      [] sinal_MP1 ?MP1; if MP1 then to direita
                         else to parado end
    end
  from direita_S1
    select
      sinal_MP1 ?MP1; if MP1 then to direita_S1
                      else to parado_S1 end
      [] sinal_MP0 ?MP0; if MP0 then to quebrado
                         else to direita_S1 end
    end
  from parado_S1
    select
      sinal_MP0 ?MP0; if MP0 then to esquerda_S1
                      else to parado_S1 end
      [] sinal_MP1 ?MP1; if MP1 then to direita_S1
                         else to parado_S1 end
    end
  from esquerda_S1
```

```
    select
      sinal_MP0 ?MP0; if MP0 then to esquerda_S1
                      else to parado_S1 end
      [] sinal_S1 !S1:=false; to esquerda
      [] sinal_MP1 ?MP1; if MP1 then to quebrado
                         else to esquerda_S1 end
   end
  from esquerda
    select
      sinal_MP0 ?MP0; if MP0 then to esquerda
                      else to parado end
      [] sinal_S0 !S0:=true; to esquerda_S0
      [] sinal_MP1 ?MP1; if MP1 then to quebrado
                         else to esquerda end
    end
  from esquerda_S0
    select
      sinal_MP0 ?MP0; if MP0 then to esquerda_S0
                      else to parado_S0 end
      [] sinal_MP1 ?MP1; if MP1 then to quebrado
                         else to esquerda_S0 end
    end
  from parado_S0
    select
      sinal_MP0 ?MP0; if MP0 then to esquerda_S0
                      else to parado_S0 end
      [] sinal_MP1 ?MP1; if MP1 then to direita_S0
                         else to parado_S0 end
    end
  from direita_S0
    select
      sinal_MP1 ?MP1; if MP1 then to direita_S0
                      else to parado_S0 end
      [] sinal_S0 !S0:=false; to direita
      [] sinal_MP0 ?MP0; if MP0 then to quebrado
                         else to direita_S0 end
    end
  from direita
    select
      sinal_MP1 ?MP1; if MP1 then to direita
                      else to parado end
      [] sinal_S1 !S1:=true; to direita_S1
      [] sinal_MP0 ?MP0; if MP0 then to quebrado
                         else to direita end
    end

/* ------------------------------------------------------------------------ */
/*       Processo que representa o botão start                              */
/* ------------------------------------------------------------------------ */
```

```
process botao_START [sinal_start: bool](START: bool) is

  states desativado, ativado
  init to desativado

  from desativado sinal_start !START:=true;
    to ativado
  from ativado sinal_start !START:=false;
    to desativado

/* ------------------------------------------------------------------------- */
/*       Processo que representa o botão stop                                */
/* ------------------------------------------------------------------------- */

process botao_STOP [sinal_stop: bool](STOP: bool) is

  states desativado, ativado
  init to desativado

  from desativado sinal_stop !STOP:=true;
    to ativado
  from ativado sinal_stop !STOP:=false;
    to desativado

/* ------------------------------------------------------------------------- */
/*   Componente que representa o main do sistema e a composição dos processos  */
/* ------------------------------------------------------------------------- */

component main is

        port
          restart:            none in [1,1],
          rung_delay:         none in [0,0],
          leitura_S0:         bool in [0,0],
          leitura_S1:         bool in [0,0],
          leitura_MR:         bool in [0,0],
          leitura_MP1:        bool in [0,0],
          leitura_MP0:        bool in [0,0],
          leitura_start:      bool in [0,0],
          leitura_stop:       bool in [0,0],
          escrita_MR:         bool in [0,0],
          escrita_S0:         bool in [0,0],
          escrita_S1:         bool in [0,0],
          escrita_MP1:        bool in [0,0],
          escrita_MP0:        bool in [0,0],
          final:              none in [0,0],
          timer1:             bool in [0,0],
          timer2:             bool in [0,0],
```

```
        timeout:            none in [3,3],
        sinal_MR:           bool in [0,0],
        sinal_start:        bool in [0,0],
        sinal_stop:         bool in [0,0],
        sinal_MP1:          bool in [0,0],
        sinal_MP0:          bool in [0,0],
        sinal_S1:           bool in [0,0],
        sinal_S0:           bool in [0,0]

  par
    par
      par
        restart, rung_delay, final, timer1, timer2, escrita_MR, escrita_MP0,
        escrita_MP1, escrita_S0, escrita_S1, leitura_S1, leitura_S0, leitura_MR,
        leitura_MP0, leitura_MP1, leitura_start, leitura_stop -> ciclo
        [restart, rung_delay, final, timer1, timer2, escrita_MR, escrita_MP0,
        escrita_MP1, escrita_S0, escrita_S1, leitura_S1, leitura_S0, leitura_MR,
        leitura_MP0, leitura_MP1, leitura_start, leitura_stop] (false,false,
        false,false,true,false,false,false,false,false,false)
        || timer, timer2, timeout -> ton [timer, timer2, timeout] (false, false)
      end
           || sinal_MR -> motor_MR [sinal_MR](false)
           || sinal_MP1, sinal_MP0, sinal_S1, sinal_S0 -> motor_MP [sinal_MP1,
           sinal_MP0, sinal_S1, sinal_S0] (false, false, false, false)
           || sinal_start -> botao_START [sinal_start](false)
           || sinal_stop -> botao_STOP [sinal_stop](false)
    end
         || leitura_S1, leitura_S0, leitura_MR, leitura_MP0, leitura_MP1,
         leitura_start, leitura_stop, sinal_S1, sinal_S0, sinal_MR, sinal_MP0,
         sinal_MP1, sinal_start, sinal_stop -> inputsGlue [leitura_S1, leitura_S0,
         leitura_MR, leitura_MP0, leitura_MP1, leitura_start, leitura_stop, sinal_S1,
         sinal_S0, sinal_MR, sinal_MP0, sinal_MP1, sinal_start, sinal_stop]
         (false,false,false,false,false,false,false)
         || escrita_MR, escrita_S0, escrita_S1, escrita_MP1, escrita_MP0, sinal_MR,
         sinal_S0, sinal_S1, sinal_MP1, sinal_MP0 -> outputsGlue [escrita_MR,
         escrita_S0,  escrita_S1, escrita_MP1, escrita_MP0, sinal_MR, sinal_S0,
         sinal_S1, sinal_MP1, sinal_MP0] (false, false, false, false, false)
  end
main
```

B.5 Resultados da ferramenta TINA para a verificação do Mixer

Nesta seção são apresentados os resultados completos da verificação por *model-checking* aplicado no exemplo do *Mixer* utilizando a

ferramenta SELT/TINA. Através do arquivo TTS gerado, é utilizada a ferramenta TINA juntamente com as fórmulas em LTL.

Primeiramente é aplicada a fórmula para verificar se o ciclo de execução é sempre completo:

```
[] (ciclo_idle => <> ciclo_fim)
Selt version 2.9.6 -- 06/10/09 -- LAAS/CNRS
ktz loaded, 232 states, 220 transitions
0.000s
TRUE
0.016s
```

Nunca vai haver estado quebrado.

```
[] -(motor_MP_quebrado)
TRUE
0.019s.
```

Na seqüência, o motor MP girando para a esquerda implica em não chegar no estado quebrado.

```
[] (motor_MP_esquerda => - motor_MP_quebrado)
TRUE
0.019s.
```

O motor MP girando para a direita implica em não chegar no estado quebrado.

```
[] (motor_MP_direita => - motor_MP_quebrado)
TRUE
0.019s.
```

Outra propriedade importante é a ordem de acionamento dos motores MR e MP, para que o sistema funcione corretamente. Após o acionamento do motor MP para a esquerda, então o motor MR acionará:

```
[] (motor_MP_esquerda => motor_MR_ligado)
TRUE
0.016s.
```

Após o acionamento do motor MP para a direita, então o motor MR não acionará:

```
[] (motor_MP_direita => - motor_MR_ligado)
TRUE
0.019s.
```

Referências Bibliográficas

[1] P. L. Antonelli. *Introdução aos Controladores Lógicos Programáveis (CLPs).* CEETPES - ETE, 1998.

[2] ATL. *Atlas Transformation Language. ATL Starter's Guide - version 0.1.* Nantes, France, 2005.

[3] M. R. A. Barros. Estudo da automação de células de manufatura para montagens e soldagem industrial de carrocerias automotivas. Master's thesis, Escola Politécnica, Universidade de São Paulo, São Paulo, Brasil, 2006.

[4] G. Behrmann, A. David, and K.G. Larsen. A tutorial on uppaal. *Formal methods for the design of real-time systems*, pages 200–236, 2004.

[5] H. C. Belan and V. J De Negri. *Bancada Didática SAP - Sistemas de Automação Pneumática: Manual de Utilização dos Módulos.* LASHIP-UFSC, Florianópolis, Brasil, 2005.

[6] D. Bender, B. Combemale, X. Cregut, J. Farines, B. Berthomieu, and F. Vernadat. Ladder metamodeling and PLC program validation through time Petri nets. In *Model Driven Architecture–Foundations and Applications*, pages 121–136. Springer, 2008.

[7] B. Berthomieu and F. Vernadat. Reseaux de Petri temporels: methodes d'analyse et verification avec TINA. *Systemes temps reel*, pages 25–58, 2006.

[8] B. Berthomieu, J. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, and F. Vernadat. *The Syntax and Semantics of FIACRE.* France, 2008.

[9] B. Berthomieu, J.P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: An intermediate language for model verification in the topcased environment. *ERTS 2008, Toulouse, France*, 2008.

[10] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems*, 14 (1):25–59, 1987.

[11] J. A. F. Bottura. Um estudo de caso de organização de programas de automação industrial baseada na norma iec61131-3. Master's thesis, Programa de Pós-Graduação Latu Sensu da Pontifícia Universidade Católica de São Paulo, São Paulo, Brasil, 2007.

[12] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and TJ Grose. Eclipse Modeling Framework (The Eclipse Series), 2003.

[13] G. Canet, S. Couffin, JJ Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *IEEE International Conference on Systems Man and Cybernetics*, volume 4, pages 2449–2454. Citeseer, 2000.

[14] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking.* Springer, 1999.

[15] P. Farail and P. Gaufillet. Topcased: un environnement de développement open source pour les systemes embarqués. *Genie logiciel(1995)*, (73):16–20, 2005.

[16] J.M. Farines, J.S. Fraga, and R.S. Oliveira. Sistemas de Tempo Real. *Florianopolis: Departamento de Automacao e Sistemas- Universidade Federal de Santa Catarina*, 2000.

[17] N. F. G. Ferreira. Verificação formal de sistemas modelados em estados finitos. Master's thesis, Escola Politécnica da Universidade de São Paulo, São Paulo, Brasil, 2005.

[18] Project FIACRE. *TOPCASED: The Open-Source Toolkit for Critical Systems*. Disponível em: http://www.topcased.org, 2009.

[19] M. Fonseca. *IEC 61131-3: a norma para programação*. Disponível em: http://www.plcopen.org, 2008.

[20] G. Frey and L. Litz. Formal methods in PLC programming. In *IEEE International Conference on Systems Man and Cybernetics*, volume 4, pages 2431–2436, Nashville, USA, 2000.

[21] G. Frey and F. Wagner. A toolbox for the development of logic controllers using Petri nets. In *Proceedings of the 8th International Workshop on Discrete Event Systems (WODES 2006), Ann Arbor, Michigan, USA*, pages 473–474, 2006.

[22] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. *Lecture Notes in Computer Science*, 4590:158, 2007.

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

[24] G. A. Guarneri. *Controladores Lógicos Programáveis - Hardware*. Universidade Tecnológica Federal do Paraná, Coordenação de Eletrônica, Campus Pato Branco, 2009.

[25] T. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. *Automata, Languages and Programming*, pages 269–279, 1987.

[26] IEC. *IEC 61131-3, 2nd Ed. Programmable Controllers - Programming Languages*, 2003.

[27] IEC. *Portal IEC 61131.* Disponível em: http://www.iec61131.com.br, 2008.

[28] S. Lohmann, L.A.D. Thi, and O. Stursberg. Design of verified logic control programs. In *2006 IEEE Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1855–1860, 2006.

[29] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 106–113, 1999.

[30] K.L. McMillan. *Symbolic model checking.* Kluwer Academic, 1993.

[31] MDA. *The Model-Driven Architecture - Guide Version 1.0.1*, 2003.

[32] H.B. Mokadem. *Vérification des propriétés temporisées des automates programmables industriels.* Thèse, École Normale Supérieure de Cachan, Cachan, France, 2006.

[33] H.B. Mokadem, B. Berard, V. Gourcuff, J.M. Roussel, and O. De Smet. Verification of a timed multitask system with Uppaal. In *ETFA*, volume 5, pages 347–354, 2005.

[34] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE control systems magazine*, 14(2):53–59, 1994.

[35] C. C. Moraes and P. L. Castrucci. *Engenharia de Automação Industrial.* 2ºed. Rio de Janeiro: Editora LTC, 2007.

[36] G. S. Nascimento. Identificação de nomes ativos em agentes-pi baseada em tipos. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil, 2005.

[37] C. H. C. Oliveira. Verificação de modelos aplicada ao projeto de sistemas industriais automatizados por controladores lógicos programáveis. Master's thesis, IME, Rio de Janeiro, Brasil, 2006.

[38] OMG. *Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Core Specification*, 2006.

[39] P. H. Pinto. *Funcionamento de um Controlador Lógico Programável (CLP).* Revista Controle de Contaminação. Disponível em: http://www.pharmaster.com.br/artigos, 2008.

[40] PLCOpen. *PLCOpen for efficiency in automation.* Disponível em: http://www.plcopen.org, 2008.

[41] O. Rossi and P. Schnoebelen. Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In *Proc. 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM 2000), Dortmund, Germany*, pages 177–182. Citeseer, 2000.

[42] R. T. Saad. Elementos para a construção de uma cadeia de verificação para o projeto topcased. Master's thesis, Universidade Federal de Santa Catarina, Florianópolis, Brasil, 2008.

[43] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer, February 2006 (Vol. 39, No. 2)*, pages 25–31, 2006.

[44] A. de M. Silva. Aplicação de verificação de modelos a programas de clp: Explorando a temporização. Master's thesis, IME, Rio de Janeiro, Brasil, 2008.

[45] M.F. Souza, J.M. Farines, and M.H. Queiroz. Modelagem e verificação de programas em Diagrama Ladder para Controladores Lógicos Programáveis. *XVIII Congresso Brasileiro de Automática - CBA 2010, Bonito-MS*, 2010.

[46] TINA. *The toolbox TINA Home Page.* Disponível em: http://www.laas.fr/tina, 2009.

[47] E. Tisserant, L. Bessard, and M. de Sousa. An open source IEC 61131-3 integrated development environment. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 1. Citeseer, 2007.

[48] PLCOpen XML. *XML Formats for IEC 61131-3*, 2005.

[49] B. Zoubek. *Automatic verification of temporal and timed properties of control programs*. Tese de doutorado, University of Birmingham, Birmingham, UK, 2004.