A reconfiguração dinâmica é um requisito chave para sistemas de automação flexíveis. Ela permite a reconfiguração do sistema em tempo de execução, tanto em hardware quanto em software. A norma IEC 61499 tem sido apontada como a solução mais adequada para a modelagem de sistemas flexíveis. Devido a sua arquitetura modular, a norma IEC 61499 permite alterar qualquer parte do software em tempo de execução. Entretanto, a sua respectiva proposta de reconfiguração dinâmica não garante a correta execução da aplicação, forçando o projetista a conhecer e verificar detalhadamente cada passo da aplicação e do processo de reconfiguração, tanto antes quanto depois deste, o que pode causar erros de controle. Esse trabalho propõe um método de reconfiguração dinâmica segura para sistemas que possam ser modelados por sistemas a eventos discretos. O método utiliza como entrada somente a configuração atualmente em execução no sistema e a nova configuração projetada. A saída é uma configuração que pode substituir a configuração atual do sistema. Essa, guiará o sistema para que antigas especificações sejam removidas e as novas acrescentadas. Dessa forma, tem-se a reconfiguração do sistema em tempo de execução, sem necessidade de o projetista determinar os passos desse processo. O método também possibilita que a reconfiguração seja executada de forma totalmente automatizada. O método desenvolvido é independente de linguagem de programação. Logo, foi demonstrada a sua aplicação em sistemas aderentes à norma IEC 61499. Para manter o processo de reconfiguração totalmente automatizado, esse trabalho demonstrou uma forma para gerar uma rede de blocos de função da IEC 61499 a partir do modelo de um sistema a eventos discretos. Dessa forma, o estudo de caso realizado pôde demonstrar a automatização do processo de reconfiguração desde a concepção da nova configuração até a sua substituição prática numa linha de produção.

Orientador: Dr. Roberto Silvio Ubertino Rosso Jr.

Coorientador: André Bittencourt Leal

Joinville, 2019

ANO
2019

LEANDRO ISRAEL PINTO

UM MÉTODO DE RECONFIGURAÇÃO DINÂMICA SEGURA E SUA APLICAÇÃO EM SISTEMAS ADERENTES À IEC 61499

UDESC

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS E TECNOLOGIA – CCT
PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA ELÉTRICA (PPGEEL)

TESE DE DOUTORADO

UM MÉTODO DE RECONFIGURAÇÃO DINÂMICA SEGURA E SUA APLICAÇÃO EM SISTEMAS ADERENTES À IEC 61499

LEANDRO ISRAEL PINTO

JOINVILLE, 2019

LEANDRO ISRAEL PINTO

UM MÉTODO DE RECONFIGURAÇÃO DINÂMICA SEGURA E SUA APLICAÇÃO EM SISTEMAS ADERENTES À IEC 61499

Documento apresentado ao Curso de Doutorado em Engenharia Elétrica, da Universidade do Estado de Santa Catarina, para exame de Qualificação.

Orientador: Prof. Dr. Roberto Silvio Ubertino Rosso Jr.
Coorientador: Prof. Dr. André Bittencourt Leal

JOINVILLE

2019

Ficha catalográfica elaborada pelo programa de geração automática da

Biblioteca Setorial do CCT/UDESC,

com os dados fornecidos pelo(a) autor(a)

Pinto, Leandro Israel
Um Método de Reconfiguração Dinâmica Segura e sua Aplicação em Sistemas Aderentes à IEC 61499 / Leandro Israel Pinto. -- 2019.
111 p.

Orientador: Roberto Silvio Ubertino Rosso Junior
Coorientador: Andre Bittencourt Leal
Tese (doutorado) -- Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Engenharia Elétrica, Joinville, 2019.

1. Reconfiguração Dinâmica. 2. Sistemas a Eventos Discretos. 3. IEC 61499. 4. Automação e Controle. 5. Controle Supervisório. I. Rosso Junior, Roberto Silvio Ubertino. II. Leal, Andre Bittencourt. III. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Engenharia Elétrica. IV. Titulo.

Um Método de Reconfiguração Dinâmica Segura e sua Aplicação em Sistemas Aderentes à IEC61499

por

Leandro Israel Pinto

Esta tese foi julgada adequada para obtenção do título de

DOUTOR EM ENGENHARIA ELÉTRICA

Área de concentração em "Sistemas Eletroeletrônicos"
e aprovada em sua forma final pelo

CURSO DE DOUTORADO EM ENGENHARIA ELÉTRICA
DO CENTRO DE CIÊNCIAS TECNOLÓGICAS DA
UNIVERSIDADE DO ESTADO DE SANTA CATARINA.

Banca Examinadora:

Prof. Dr. Roberto Silvio Ubertino Rosso Junior
CCT/UDESC (Orientador/Presidente)

Prof. Dr. Cristiano Damiani Vasconcellos
CCT/UDESC

Prof. Dr. Carlos Eduardo Pereira
UFRGS

Prof. Dr. Douglas Wildgrube Bertol
CCT/UDESC

Professora Mariana Santos Matos Cavalca
Coordenadora do PPGEE
UDESC - Joinville
p/ Prof. Dr. Sergio Vale Aguiar Campos
UFMG
(Por videoconferência)

Joinville,SC, 29 de agosto de 2019.

Dedico este trabalho aos meus familiares, amigos, colegas e professores que me acompanharam e me deram forças nessa magnífica trajetória.

RESUMO

A reconfiguração dinâmica é um requisito chave para sistemas de automação flexíveis. Ela permite a reconfiguração do sistema em tempo de execução, tanto em hardware quanto em software. A norma IEC 61499 tem sido apontada como a solução mais adequada para a modelagem de sistemas flexíveis. Devido a sua arquitetura modular, a norma IEC 61499 permite alterar qualquer parte do software em tempo de execução. Entretanto, a sua respectiva proposta de reconfiguração dinâmica não garante a correta execução da aplicação, forçando o projetista a conhecer e verificar detalhadamente cada passo da aplicação e do processo de reconfiguração, tanto antes quanto depois deste, o que pode causar erros de controle.

Esse trabalho propõe um método de reconfiguração dinâmica segura para sistemas que possam ser modelados por sistemas a eventos discretos. O método utiliza como entrada somente a configuração atualmente em execução no sistema e a nova configuração projetada. A saída é uma configuração que pode substituir a configuração atual do sistema. Essa, guiará o sistema para que antigas especificações sejam removidas e as novas acrescentadas. Dessa forma, tem-se a reconfiguração do sistema em tempo de execução, sem necessidade de o projetista determinar os passos desse processo. O método também possibilita que a reconfiguração seja executada de forma totalmente automatizada. O método desenvolvido é independente de linguagem de programação. Logo, foi demonstrada a sua aplicação em sistemas aderentes à norma IEC 61499. Para manter o processo de reconfiguração totalmente automatizado, esse trabalho demonstrou uma forma para gerar uma rede de blocos de função da IEC 61499 a partir do modelo de um sistema a eventos discretos. Dessa forma, o estudo de caso realizado pôde demonstrar a automatização do processo de reconfiguração desde a concepção da nova configuração até a sua substituição prática numa linha de produção.

Palavras-chaves: Reconfiguração Dinâmica, Sistemas a Eventos Discretos, IEC 61499 e Automação e Controle.

ABSTRACT

Dynamic reconfiguration is a key requirement for flexible automation systems. It allows runtime system reconfiguration both in hardware and software. The IEC 61499 standard has been cited as the most suitable solution for modeling flexible systems. Due to its modular architecture, IEC 61499 allows to change any part of the software at runtime. However, their proposal for dynamic reconfiguration does not guarantee the correct execution of the application, forcing the designer to know and verify in details each step of the reconfiguration process, which can cause control errors.

This work proposes a safe dynamic reconfiguration method for systems that can be modeled by discrete event systems. The method uses only the current system configuration and the new designed configuration as input. The output is a configuration that can override the current one. This will guide the system to remove old specifications and add new ones. Thus, the system is reconfigured at runtime, without the designer having to determine the steps of this process. The method also enabled the reconfiguration to be performed in a fully automated manner. The proposed method is independent of programming language. Therefore, its application in IEC 61499 compliant systems has been demonstrated. To maintain a fully automated reconfiguration process, this work has demonstrated a way to generate an IEC 61499 function block network from the discrete event system model. In this way, the case study performed could demonstrate the automation of the reconfiguration process from the conception of the new configuration until its practical replacement in a production line.

Key-words: Dynamic Reconfiguration, Discrete Event Systems, IEC 61499 and Automation and Control.

LISTA DE ILUSTRAÇÕES

Figura 1 – Modelo de Sistema da IEC 61499. 29
Figura 2 – Modelo de Dispositivo da IEC 61499. 30
Figura 3 – Representação gráfica de um bloco de funções. 31
Figura 4 – Um bloco de funções básico. 32
Figura 5 – ECC de exemplo. 33
Figura 6 – Um bloco de funções do tipo TMP_CONTROL. 33
Figura 7 – ECC do bloco TMP_CONTROL a esquerda e os algoritmos internos do bloco a direita. 34
Figura 8 – Definição do SIFB IO_READER. 36
Figura 9 – Um *Composite function block*. 36
Figura 10 – Um exemplo de uma FBN para controle de temperatura. 37
Figura 11 – Um dispositivo com aplicação de gerenciamento no Recurso MGR. 38
Figura 12 – Um exemplo de comandos de gerenciamento criando uma rede de blocos de função. 41
Figura 13 – Representação gráfica do autômato G. 46
Figura 14 – Planta sequencial. 47
Figura 15 – Autômato para controle da planta na Figura 14. 47
Figura 16 – Controle em malha fechada. 48
Figura 17 – Autômato para a máquina m_i. 14. 49
Figura 18 – Modelo de planta G que representa o comportamento fisicamente possível. . 50
Figura 19 – Especificações para a planta da Figura 14. 51
Figura 20 – Supervisor para a planta da Figura 14. 52
Figura 21 – Processo de configuração estática em (a) e dinâmica em (b) 56
Figura 22 – Planta com as duas máquina $m1$ e $m2$. A máquina $m2$ é inserida no processo por meio de uma RD. 61
Figura 23 – (a) mostra C_1 com apenas a máquina $m1$ e (b) mostra C_2 com a nova máquina $m2$ operando antes de $m1$. 62
Figura 24 – A LC $C_{1\rightarrow 2}$, que é o retorno do Algoritmo Rutt ao receber C_1 e C_2 como entrada. 63
Figura 25 – Fluxo de execução da RDS. 63
Figura 26 – Resultado do algoritmo $ISOTRIM$. Somente os estados 4 e 5 e o evento d são removidos do autômato original. 65
Figura 27 – Em (a) a LC inicial do sistema. Em (b) a nova LC. 68
Figura 28 – Processamento de cada parte do algoritmo sobre as LCs do sistema de semáforos. 69

Figura 29 – Controlador de atualização para a célula de produção com três fases. C_G e $C_{G'}$ garantem as especificações antigas e novas respectivamente. Os estados intermediários garantem o progresso da atualização. 72
Figura 30 – LC anterior e nova LC para a célula de produção de Nahabedian et al. (2018). 73
Figura 31 – LC gerada pelo algoritmo $Rutt$ para a célula de produção de Nahabedian et al. (2018). 73
Figura 32 – Planta do SMR com caminho paralelo. 74
Figura 33 – LC C_1 para a planta com caminho paralelo. 75
Figura 34 – LC $C_{1\to2}$ para a planta com caminho paralelo. 76
Figura 35 – (a) Uma célula de produção (b) A mesma célula de produção com a máquina $m2$ substituída por $m2n$. 77
Figura 36 – LC C_4. 77
Figura 37 – LC C_5. 78
Figura 38 – Modelo $C_{4\to5}$ 79
Figura 39 – Planta didática utilizada para este estudo de caso. 82
Figura 40 – Exemplo de uma estrutura geral que implementa o método RDS num SMR aderente a IC 61499. 83
Figura 41 – Uma representação genérica de uma FBN para implementar o modelo de um controlador. 84
Figura 42 – O ECC de um FB SUP_Cx. 85
Figura 43 – Exemplo de uma LC. 86
Figura 44 – Representação geral da ARD. 88
Figura 45 – Interface gráfica da ARD. 90
Figura 46 – O ECC do FB AUTODR. 91
Figura 47 – Planta didática utilizada para este estudo de caso. No canto superior direito, a planta equivalente na simulação. 94
Figura 48 – Uma representação genérica de uma FBN para implementar o modelo de um controlador. 107
Figura 49 – Interface do FB AUTODR. 113

LISTA DE TABELAS

Tabela 1 – Entradas e saídas padronizadas para SIFBs 35
Tabela 2 – Função de transição δ do Autômato G. 45
Tabela 3 – Abordagens de Modelagem Formal para a IEC 61499. 57

LISTA DE SIGLAS E ABREVIATURAS

AFD Autômato Finito Determinístico

ARD Aplicação de Reconfiguração Dinâmica

CFB *Composite Function Block*

CNet Component Net

DCS *Distributed Control System*

FB *Function Block*

FBD *Function Block Diagram*

FBN *Function Block Network*

FSM Finite State Machine

IEC *International Electrotechnical Commission*

IEC *Service Interface Function Block*

IL *Instruction List*

LC Lógica de Controle

NCES Net Condition Event System

RD Reconfiguração Dinâmica

RDS Reconfiguração Dinâmica Segura

RFB *Reconfigurable Function Block*

SED Sistema a Eventos Discretos

SFC *Sequential Function Chart*

SMR Sistema de Manufatura Reconfigurável

ST *Structured Text*

TCS Teoria de Controle Supervisório

TM Transition System

*

SUMÁRIO

1 INTRODUÇÃO 21
1.1 OBJETIVO 25
1.2 ORGANIZAÇÃO DO TRABALHO 25

2 A NORMA IEC 61499 27
2.1 MODELO DE SISTEMA 29
2.2 MODELO DE DISPOSITIVO 30
2.3 MODELO DE RECURSO 30
2.4 MODELO DE APLICAÇÃO 31
2.5 BLOCO DE FUNÇÃO 31
2.5.1 Bloco de Funções Básico 32
2.5.2 Bloco de Funções de Interface de Serviço 34
2.5.3 Bloco de Funções Composto 36
2.5.4 Rede de Blocos de Função 36
2.6 MODELO DE GERENCIAMENTO 37
2.7 AMBIENTES DE EXECUÇÃO ADERENTES A IEC 61499 38
2.8 RECONFIGURAÇÃO DINÂMICA 41
2.9 CONSIDERAÇÕES DO CAPÍTULO 42

3 CONTROLE SUPERVISÓRIO DE SISTEMAS A EVENTOS DISCRETOS . . 43
3.1 SISTEMAS A EVENTOS DISCRETOS 43
3.2 LINGUAGENS 44
3.3 AUTÔMATO FINITO DETERMINÍSTICO 45
3.4 EXEMPLO DE UM SED 46
3.5 CONTROLE SUPERVISÓRIO DE SEDS 48
3.6 CONSIDERAÇÕES DO CAPÍTULO 52

4 TRABALHOS RELACIONADOS 55
4.1 ANÁLISE DOS TRABALHOS RELACIONADOS 56
4.2 CONSIDERAÇÕES DO CAPÍTULO 59

5 RECONFIGURAÇÃO DINÂMICA SEGURA 61
5.1 GERANDO O MODELO $C_{1\to 2}$ 64
5.1.1 Algoritmo ISOTRIM 64
5.1.2 O Algoritmo *Rutt* 65
5.1.3 Definição de Estado Equivalente 67

5.1.4 Exemplo de Execução do Algoritmo Rutt 67
5.2 Exemplo de uma célula de Produção 70
5.3 Exemplo de Uma Célula de Produção com Caminho Paralelo . . . 74
5.4 Exemplo de Uma Célula de Produção com Descarte 76
5.5 Considerações do Capítulo 78

6 AUTOMATIZANDO A RDS EM SISTEMAS ADERENTES À IEC 61499 . . . 81
6.1 Estrutura Aderente a IEC 61499 para Automatização da RDS . . 81
6.2 Arquitetura da FBN 82
6.3 FB Gerado a Partir do Modelo de um Supervisor 84
6.4 Aplicação de Reconfiguração Dinâmica 88
6.5 Gerando a ARD automaticamente 90
6.6 Considerações do Capítulo 94

7 CONCLUSÕES E TRABALHOS FUTUROS 97
7.1 Limitações e Trabalhos Futuros 98

REFERÊNCIAS 101

APÊNDICE A BLOCO DE FUNÇÃO DA LINHA DE PRODUÇÃO 107

APÊNDICE B BLOCO DE FUNÇÃO AUTODR 113

1 INTRODUÇÃO

A reconfiguração dinâmica (RD) tem se tornado um fator chave para o sucesso de sistemas de manufatura reconfiguráveis (SMR). Sistemas de manufatura em geral focam na produção em massa de produtos homogêneos. Porém, com as novas demandas de mercado, como o curto ciclo de vida dos produtos e customizações, a flexibilidade nestes sistemas tem sido cada vez mais requisitada. Um mecanismo, que permita a adaptação eficiente a novos produtos, pode garantir a fabricação de produtos customizados com velocidade semelhante à produção de produtos padronizados. (ZOITL et al., 2005; ALSAFI; VYATKIN, 2010; ATMOJO; SALCIC; WANG, 2018; ATMOJO et al., 2018). Um exemplo é mostrado por Vyatkin (2011), no qual uma linha de produção pode produzir sapatos customizados com velocidade de produção em massa.

A RD é a habilidade de modificar o software e hardware em tempo de execução. Neste trabalho, o foco é a aplicação da RD num sistema de manufatura. Outros sistemas como, por exemplo, de automação residencial ou predial, fornecimento de energia, monitoramento e alarmes são exemplos de sistemas que podem se beneficiar da RD. Pois são sistemas que nunca devem ser desligados. A reconfiguração tradicional de um sistema de manufatura se dá através do completo desligamento do mesmo, carregamento da nova lógica de controle (LC), alterações de hardware e reinício do sistema. Isso pode significar longos períodos de inatividade gerando prejuízos financeiros (PRENZEL; PROVOST, 2017).

Kramer e Magee (1985) identificaram as características da reconfiguração tradicional e dinâmica. Eles também identificaram os requisitos necessários para suportar a RD. Um desses requisitos é que o controlador do sistema de manufatura deve fornecer recursos que possibilite a troca do software em tempo de execução.

A norma IEC 61499-1 (2012) apresenta as características necessárias para a RD. Ela é uma evolução da norma IEC 61131-3 (2003), que tem sido amplamente utilizada na implementação de sistemas de manufatura. A IEC 61131-3 fornece as linguagens como LADDER, Diagrama de Bloco de Função (FBD), Texto Estruturado (ST), Lista de Instruções (IL) e *Sequential Function Chart* (SFC). Enquanto que a norma IEC 61499 fornece uma arquitetura, baseada em blocos de função para a modelagem de um sistema de automação e controle distribuído, sendo que o bloco de função da IEC 61499 é denotado por FB nesta tese. Deve-se denotar aqui que os FBs são uma evolução dos blocos de função da norma IEC 61131-3. A norma IEC 61499 adota os eventos como forma de controlar o fluxo de execução dos FBs, enquanto a IEC 61131 faz uma execução sequencial de todos os blocos. A parte 4 da norma (IEC 61499-4, 2013) fornece os comandos de reconfiguração, os quais permitem a modificação dos FBs em tempo de execução.

Entretanto, os comandos de reconfiguração, apresentados na IEC 61499-4 (2013), não garantem que um SMR continue operando corretamente (ALMEIDA; LUNTZ; TILBURY, 2007; Guellouz et al., 2016). Alguns trabalhos, discutidos no Capítulo 4, mostram que, para executar a RD, o sistema de manufatura deve ser conduzido adequadamente de sua configuração atual até a nova, o que exige conhecimentos detalhados de ambas as configurações. Nesse processo, devem ser tratados inúmeros problemas como: o que fazer com os produtos que já estão na linha de produção e em diferentes estágios de construção? Em se tratando da troca de uma máquina, como garantir que a mesma não seja mais requisitada pelo sistema durante a troca? Se a segunda máquina possuir mais funcionalidades que a primeira, o que deve ser feito com a LC? Como a LC deve controlar o sistema desde a sua configuração atual até a nova? Ou seja, como eliminar com segurança uma antiga especificação e/ou acrescentar uma nova?

Não há uma técnica de verificação formal para garantir que o processo de RD, projetado manualmente, irá funcionar num sistema de manufatura. Isso se deve ao fato de não existir um formalismo capaz de modelar completamente um sistema desenvolvido com a norma IEC 61499 para, então, realizar uma verificação formal. Os trabalhos dessa área modelam parcialmente os FBs e, em alguns, os autores indicam que é impossível a modelagem total da norma através dos formalismos conhecidos. Esses trabalhos serão apresentados no Capítulo 4.

Na área de sistemas a eventos discretos (SEDs) é mais comum encontrar formalismos matemáticos. Em geral, pode-se modelar um SED através de um autômato ou uma rede de Petri. Um SED é um sistema dinâmico que evolui de acordo com a ocorrência de eventos físicos em intervalos irregulares e possivelmente desconhecidos. Por exemplo, um evento pode indicar o fim de operação de uma máquina numa linha de manufatura ou a chegada de peças numa determinada posição através de uma esteira (CASSANDRAS; LAFORTUNE, 2008).

Mesmo sistemas de natureza puramente contínua podem ser modelados como um SED. Uma forma conhecida de fazer isso é atribuir valores limites para uma variável contínua, abstraindo esse comportamento, e então, discretizar essas informações. Por exemplo, para o controle de temperatura pode-se associar valores de temperatura mínima e máxima desejada e, então, disparar um evento quando a temperatura atingir o valor mínimo e outro quando atingir o valor máximo (LEAL; CURY, 2004).

Na área de SEDs, a RD pode ser referenciada como *Dynamic Software Update*, *on-the-fly program modification*, *hot-swapping* (SEIFZADEH; ABOLHASSANI; MOSHKENANI, 2013). Porém, segundo (NAHABEDIAN et al., 2018), vários dos trabalhos nessa área assumem que a reconfiguração a ser feita é conhecida durante o projeto. Em contraste, esta tese não assume que as novas especificações são conhecidas. O trabalho de Nahabedian et al. (2018) atende a este requisito e é o trabalho encontrado que mais se aproxima desta tese. O método proposto permite conduzir o sistema de uma configuração para outra, permitindo-o a atender requisitos não especificados anteriormente. Porém, para esta condução é gerado um grande número de estados extras. O método também não considera o período de

tempo necessário para modificar o sistema fisicamente, por exemplo, trocar uma máquina exige intervenção de pessoas por algum período. O método proposto nesta tese indica quais eventos devem ser temporariamente desabilitados para possibilitar essas intervenções no SMR. Além de não gerar estados extras para o processo de RD.

Segundo Seifzadeh, Abolhassani e Moshkenani (2013), muitos sistemas que utilizam a reconfiguração dinâmica ignoram a limpeza de código. A alteração do software em geral exige mais código do que o utilizado somente para controle. Esse código extra em geral executa as tarefas para executar a reconfiguração, ao fim desta, esse código pode ser descartado. O método proposto nesta tese indica quais estados e transições podem ser removidos com segurança ao término da RD. Estes estados e transições representam especificações e eventos que se tornaram obsoletos após a RD.

Nesta tese, é proposto um método de reconfiguração dinâmica segura (RDS) para aplicação em SEDs, o qual foi descrito inicialmente em Pinto, Leal e Rosso (2017). O método se baseia no fato de que sempre há um estado equivalente entre duas configurações para um mesmo SMR, o qual é o estado inicial e, por este motivo, é seguro realizar qualquer alteração. Uma vez que, no estado inicial, não há nenhum processo do SMR em execução.

A partir da afirmação de que o estado inicial é sempre seguro para a RD, foi desenvolvido um algoritmo, que é a parte principal do método, para encontrar todos os demais possíveis estados equivalentes entre duas LCs. O algoritmo considera ainda os estados não equivalentes e gera uma LC que respeite e remova os requisitos obsoletos e adicione os novos. Quanto menor a diferença entre as duas LCs, mais estados equivalentes podem ser encontrados.

A RDS reutiliza o conceito de planta e eventos controláveis e não controláveis da Teoria de Controle Supervisório para SEDs (TCS) (CASSANDRAS; LAFORTUNE, 2008). O termo planta se refere à configuração física do SMR sem nenhum tipo de controle, a qual pode ser composta por elementos como máquinas, robôs e outros. A planta não é capaz de executar tarefas, ela necessita de uma LC para realizar o controle. Na TCS, a LC é chamada supervisor, o qual é gerado através de um processo sistemático. Nesta tese, utiliza-se o termo LC para evidenciar que a mesma não precisa ser um supervisor, basta que esteja correta. A vantagem do uso da TCS para gerar uma LC é o fato de que um supervisor representa o controle ótimo e minimamente restritivo da planta. Como consequência, a RDS consegue identificar o maior número de estados equivalentes possível, o que representa mais tarefas que podem ser executadas durante o processo de RD.

A TCS divide os eventos em dois grupos: os controláveis e não controláveis. Isso é essencial para que a RDS permita as modificações físicas no SMR, enquanto que o restante do mesmo permanece em execução. Geralmente, associam-se os eventos controláveis a aqueles que podem ser disparados, como por exemplo, solicitar que uma máquina inicie sua operação. Os não controláveis são eventos que devem ser lidos como, por exemplo, o fim de operação de uma

máquina. A RDS desabilita temporariamente o disparo de eventos controláveis que iniciam, por exemplo, uma operação numa máquina que ainda está sendo instalada no SMR. A reabilitação deste evento é feita quando é indicado o término da instalação.

Uma comparação foi realizada entre os resultados de Nahabedian et al. (2018) e os desta tese, a qual mostrou as diferenças da condução de uma configuração para outra dos dois métodos. Foi utilizado como estudo de caso a mesma célula de produção apresentada por Nahabedian et al. (2018).

Também foi solucionado um problema de RD, apresentado por Pinto et al. (2016), de um sistema de semáforos. O estudo de caso consiste em dois semáforos para veículos (com luzes verde, amarela e vermelha) e um para pedestres (com luzes verde e vermelha). A LC de controle foi modelada com a IEC 61499 e executada pela plataforma ICARU-FB. O problema e a solução são mostrados num estudo de caso similar desta tese no Capítulo 5.

O método proposto nesta tese foi validado em outros dois estudos de caso. O primeiro, executado numa simulação, demonstrou como o SMR continua operando durante a substituição de uma de suas máquinas. O segundo foi executado tanto numa simulação quanto num SMR didático, este demonstrou a aplicação do método com a IEC 61499.

Como a norma IEC 61499 é considerada promissora para RD em SMR, já surgiu no mercado um considerável número de dispositivos compatíveis com a IEC 61499 (Eclipse 4DIAC, 2019; Rockwell Automation, 2019; LOYTEC, 2019). Assim, esta tese também propõe um método de conversão de um SED em uma rede de FBs. Uma vez que a LC é modelada por autômatos, foi desenvolvido um algoritmo capaz de converter a LC num FB. Em seguida, foi desenvolvida uma estrutura padronizada de rede de FBs para dar suporte à execução do FB gerado. Para alterar essa rede em tempo de execução, foi também desenvolvida uma aplicação de reconfiguração dinâmica (ARD), a qual é carregada para o SMR em tempo de execução para realizar a RD. Um FB da ARD é o responsável por coordenar a RD, e sua implementação interna muda a cada processo de RD. Entretanto, devido as padronizações, este FB pode ser gerado automaticamente. Foi demonstrado como o método proposto de RD pode coordenar a reconfiguração de SMR modelado pela IEC 61499.

Os estudos de caso demonstraram as vantagens do método proposto como: executar a uma RD sem precisar especificar como o SMR deve ser conduzido para a nova configuração; tratamento de situações práticas como, por exemplo, fornecer tempo para alterações físicas do sistema; possibilidade do projetista trabalhar somente no nível de modelo como, por exemplo, na aplicação à IEC 61499 alteram-se somente os modelos, enquanto que a implementação é gerada automaticamente assim como os processos de RDs.

1.1 OBJETIVO

O objetivo desta tese é desenvolver um método de reconfiguração dinâmica seguro e aplicá-lo em sistemas aderentes à IEC 61499.

Como objetivos específicos tem-se: (a) Estudar os atuais métodos de reconfiguração dinâmica em sistema aderentes a IEC 61499; (b) Estudar os atuais métodos de reconfiguração dinâmica em SEDs; (c) Desenvolver o método de reconfiguração dinâmica seguro; (d) Demonstrar o funcionamento do método em SEDs; (e) Demonstrar o funcionamento do método em aplicações aderentes a IEC 61499; (f) desenvolver um método de transformação de um SED num bloco de função da IEC 61499 e (g) desenvolver um processo automatizado para executar a RD nessas aplicações.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado como segue:

O Capítulo 2 apresenta a norma IEC 61499. Nesse Capítulo também são mostradas as características das diferentes implementações dos ambientes de execução atuais aderentes à norma IEC 61499. No Capítulo 3 é apresentada a teoria do controle supervisório, a partir da qual foi gerada a lógica de controle para os sistemas testados nesse trabalho. No Capítulo 4 são apresentados os trabalhos relacionados à reconfiguração dinâmica em sistemas aderentes a norma IEC 61499, assim como, os vários métodos de modelagem propostos para a mesma e as dificuldades encontradas na norma.

O Capítulo 5 trata o tema central deste trabalho, nele é apresentada a RDS e aplicada em sistemas a eventos discretos. Dois estudos de caso são apresentados para validação da proposta. O primeiro faz uma comparação com um trabalho similar. E o segundo tem como objetivo demonstrar que o SMR se mantém em execução correta mesmo com uma alteração física em andamento pelo projetista.

No Capítulo 6 é tratada a segunda parte do tema deste trabalho, que é a aplicação da reconfiguração dinâmica em sistemas aderentes a norma IEC 61499. Esse capítulo mostra os problemas inerentes a IEC 61499 e as soluções para contorná-los. Foi possível demostrar como o projetista pode lidar com o problema de RD somente no nível de modelo. Por exemplo, foram realizadas alterações nas especificações e na configuração física do sistema, diante dessa situação, a RDS mostrou-se eficaz em conduzir o sistema da antiga configuração para a nova sem necessitar de intervenções manuais. Finalmente, o Capítulo 7 apresenta as conclusões finais deste trabalho.

2 A NORMA IEC 61499

A Comissão Eletrotécnica Internacional (IEC - *International Electro-technical Commission*), com base em Genebra, formula os padrões internacionais para equipamentos elétricos e eletrônicos. Esses padrões são adotados na Europa e em muitos outros países, sendo que alguns incorporam tais padrões num versão nacionalizada.

A Norma IEC 61499 (IEC 61499-1, 2012), surgiu com o objetivo de padronizar o desenvolvimento de sistemas de automação distribuídos. Ela define como os blocos de funções (FB - *Function Block*) podem ser usados em processos industriais distribuídos, monitoramento e sistemas de controle. Esses FBs são utilizados como uma linguagem de programação visual para sistemas de automação e controle. Entretanto, a norma define a linguagem visual como uma modelagem para estes sistemas, e não como uma linguagem de programação.

A norma IEC 61499 estende o conceito de blocos de função definidos na norma IEC 61131-3 (IEC 61131-3, 2003), tornando-os mais adequados para o desenvolvimento de sistemas distribuídos. Uma das vantagens da utilização dos blocos de funções é a redução da complexidade de programação de sistemas de controle distribuídos (DCS) (CHOUINARD; BRENNAN, 2006; ZOITL et al., 2005; VYATKIN, 2009). O conceito de blocos de funções é utilizado para descrever um DCS inteiro, independente da plataforma de execução (YOONG; ROOP; SALCIC, 2009).

O principal objetivo da norma IEC 61499 é definir um modelo capaz de atender aos requisitos de portabilidade, configurabilidade e interoperabilidade, definidos como (STRASSER et al., 2008; LEWIS, 2008):

Portabilidade: A capacidade de executar o mesmo código em diferentes dispositivos. E ainda, alterar o mesmo código em diferentes ferramentas *software*, de maneira independente de fabricante;

Configurabilidade: A capacidade de configurar (ou programar) diferentes dispositivos, inclusive de diferentes marcas, a partir de uma única ferramenta de *software*;

Interoperabilidade : A capacidade dos dispositivos, independente de fabricante, de operarem juntos a fim de executar as tarefas de um determinado sistema.

Apesar das normas anteriores padronizarem as linguagens de programação, tanto textuais quanto gráficas, muitos CLPs e ferramentas de *software* são incompatíveis entre si, principalmente quando são de diferentes fabricantes. Assim, além dos requisitos citados, a norma IEC 61499 define também um formato *xml* padronizado para armazenar a definição dos

sistemas em um ou mais arquivos (IEC 61499-4, 2013). Isso permite que qualquer ferramenta de *software* seja capaz de editar um sistema aderente a norma IEC 61499.

Um FB tem como principal característica a capacidade de encapsular uma solução para um pequeno problema, o qual faz parte de um sistema. O FB é uma abstração de código capaz de executar em qualquer plataforma compatível com a IEC 61499 (VYATKIN, 2012). Existem três tipos de FBs na norma IEC 61499, que são:

- Bloco de Funções Básico ou *Function Block* (FB);
- Bloco de Funções Composto ou *Composite Function Block* (CFB);
- Bloco de Funções de Interface de Serviços ou *Service Interface Function Block* (SIFB).

A norma IEC 61499 pode modelar um sistema em vários níveis de abstração. O modelo com maior nível de abstração é o modelo de Sistema, no qual se tem a visualização dos dispositivos presentes no sistema. Assim, os modelos definidos pela IEC 61499 são (IEC 61499-1, 2012):

- Modelo de Sistema;
- Modelo de Dispositivo;
- Modelo de Recurso;
- Modelo de Aplicação;
- Modelo de Bloco de Funções (que podem ser de três tipos: FB, CFB e SIFB);
- Modelo de Gerenciamento;
- Modelo de Estado Operacional.

As Seções 2.1 até 2.6 descrevem os modelos de acordo com a norma IEC 61499-1 (IEC 61499-1, 2012). A norma IEC 61499 está em desenvolvimento e ainda não é amplamente aceita na indústria. Além disso, a norma contém problemas de ambiguidade nas suas definições (FERRARINI; VEBER, 2004; VYATKIN, 2009; SUNDER et al., 2006; STRASSER et al., 2011; CENGIC; AKESSON, 2010; DUBININ; VYATKIN; SHALYTO, 2016). A norma é composta por 4 partes.

A parte 1 (*Part 1: Architecture*) contém os requisitos gerais da norma, descrição de todos os modelos, regras para o uso de blocos de funções e regras para o uso de blocos de funções no gerenciamento de aplicações. O modelo de gerenciamento de aplicações será mostrado na Seção 2.6 (IEC 61499-1, 2012).

A parte 2 (*Part 2: Software tool requirements*) define os requisitos para que ferramentas de *software* suportem as tarefas de engenharia como especificar tipos de blocos de funções, configuração, implementação, operação e manutenção de sistemas distribuídos e exportação de informações entre ferramentas de *software* (IEC 61499-2, 2012).

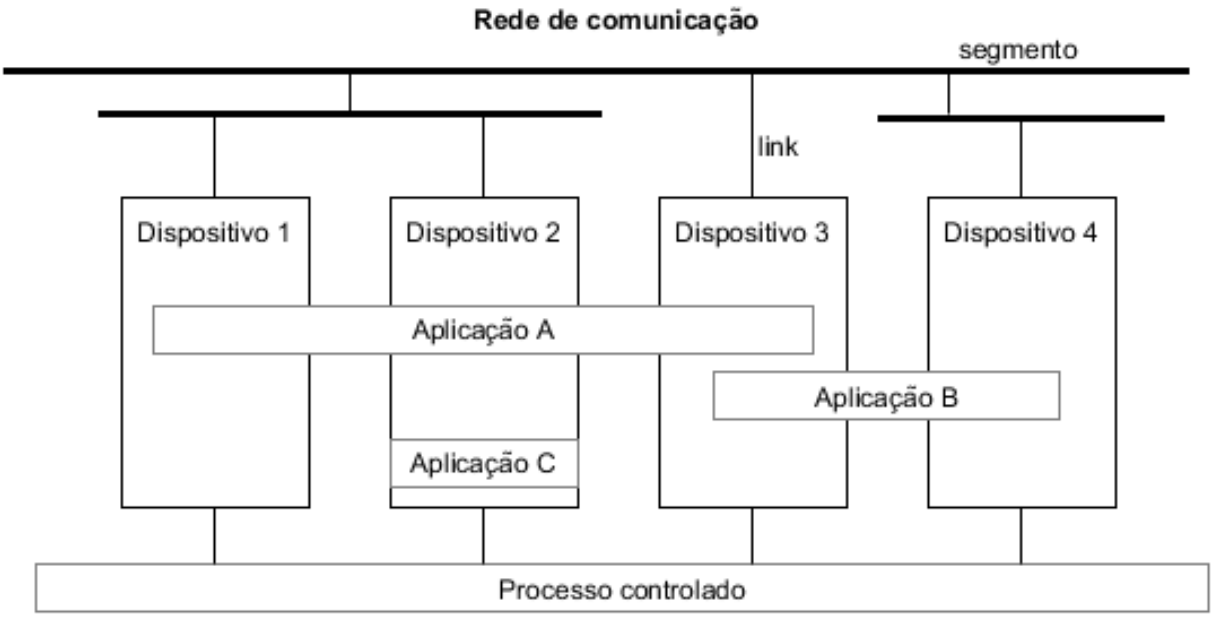
A parte 3 (*Part 3: Tutorial information*), a qual ainda está em desenvolvimento, conterá materiais educacionais em relação a IEC 61499.

A parte 4 (*Part 4: Rules for compliance profiles*) define regras para o desenvolvimento de perfis de conformidade, os quais especificam os recursos a serem implementados a fim de promover a interoperabilidade, portabilidade e configurabilidade (IEC 61499-4, 2013).

2.1 MODELO DE SISTEMA

No contexto da IEC 61499, um sistema é modelado como um conjunto de dispositivos interconectados por uma rede de comunicação com segmentos e *links*. Dispositivos são conectados a um segmento de rede via *links*. A Figura 1 mostra o modelo de um sistema.

Figura 1 – Modelo de Sistema da IEC 61499.



Fonte: Adaptado de (IEC 61499-1, 2012).

Os dispositivos do sistema cooperam a fim de executar diversas aplicações que fazem parte do sistema. Uma aplicação controla uma parte do sistema, e geralmente, requer a interoperação de mais de um dispositivo. Assim, uma aplicação pode existir em um único dispositivo ou ser distribuída em mais de um dispositivo, como as aplicações A, B e C da Figura 1.

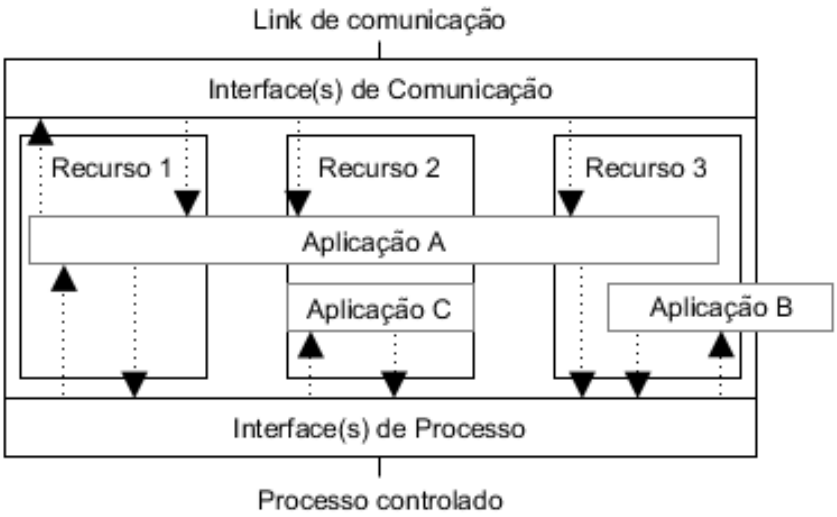
2.2 MODELO DE DISPOSITIVO

O modelo de Dispositivo, mostrado na Figura 2, representa um dispositivo físico que faz parte do sistema. Um dispositivo contém uma ou mais interfaces de comunicação e de processo e um ou mais Recursos.

A interface de comunicação é o acesso às tecnologias de comunicação presentes no dispositivo. A interface de processo é o acesso às entradas e saídas do dispositivo, as quais são utilizadas para leituras de sensores e controle de acionadores.

Um dispositivo é responsável por dar suporte a execução dos Recursos, os quais são responsáveis por executar redes de FBs. Uma aplicação pode ser distribuída entre vários Recursos, incluindo Recursos de outros dispositivos, como é o caso da Aplicação B na Figura 2.

Figura 2 – Modelo de Dispositivo da IEC 61499.



Fonte: Adaptado de IEC 61499-1 (2012)

O Dispositivo fornece ao recurso o acesso as interfaces de comunicação e de processo. O Recurso fornece esse acesso aos FBs contidos nele. Cada Recurso funciona com uma execução independente dos demais, não sendo possível a interferência de um no outro.

2.3 MODELO DE RECURSO

Segundo a norma IEC 61499, um Recurso é uma unidade funcional com controle independente sobre sua operação, e está contida num dispositivo. Ele pode ser criado, configurado, parametrizado, iniciado e excluído sem afetar outros Recursos.

O Recurso fornece suporte a execução de um ou mais FBs, os quais podem pertencer a uma ou mais aplicações. Ele fornece acesso aos recursos específicos do dispositivo, como

acesso ao hardware ou ao processo e recursos para o processamento de FBs. Através das funcionalidades fornecidas pelo Recurso os FBs podem acessar, por exemplo, sensores e atuadores, bem como trocar informações na rede de comunicação.

2.4 MODELO DE APLICAÇÃO

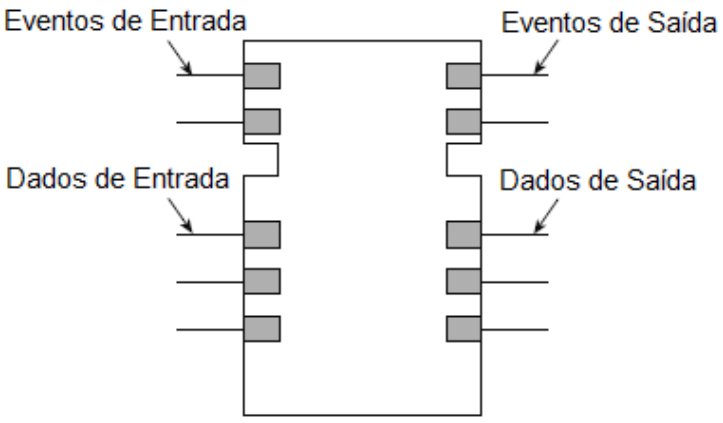
Uma aplicação IEC 61499 é definida como uma rede de FBs interconectados. Uma aplicação pode ser distribuída entre vários Recursos que podem estar em um ou em vários Dispositivos. Uma aplicação proporciona uma solução para um problema particular de um sistema de automação e controle.

2.5 BLOCO DE FUNÇÃO

O bloco de função (FB) é o elemento central da norma IEC 61499. Todas as aplicações são modeladas a partir de redes de FBs. Eles possibilitam a distribuição da aplicação, o reuso e a portabilidade de código.

A Figura 3 ilustra um FB. A parte superior contém eventos de entrada e de saída, enquanto que a inferior contém dados de entrada e de saída. O FB possui um identificador de tipo e pode ser instanciado numa determinada aplicação.

Figura 3 – Representação gráfica de um bloco de funções.



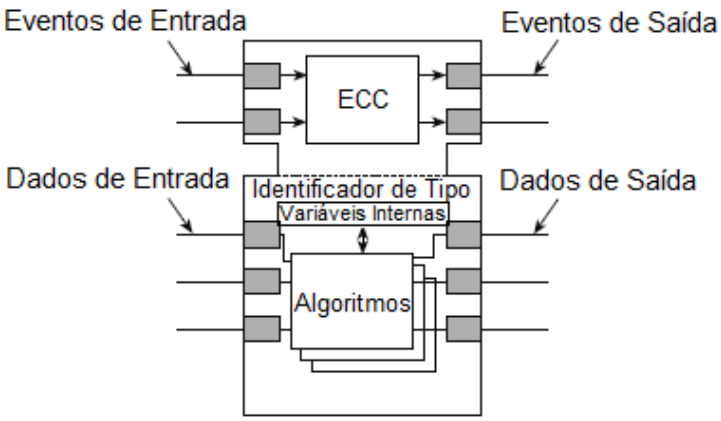
Fonte: Adaptado de Lewis (2008).

Uma característica do FB da norma IEC 61499, que os difere dos blocos de função de outras normas, é o fato de sua execução ser disparada somente quando ocorre um determinado evento de entrada. Uma rede de FB (ou uma aplicação) consiste num conjunto de FBs cujos eventos e variáveis estão interconectados, assim, o disparo de um evento de saída pode disparar a execução de outro bloco de funções, causando um fluxo de execução.

2.5.1 Bloco de Funções Básico

Um bloco de funções básico (BFB) é um FB que contém internamente um ECC (*Execution Control Chart*), variáveis internas e algoritmos. A Figura 4 mostra o modelo desse bloco. Esse tipo de bloco pode ser criado pelo usuário a fim de atender as necessidades da aplicação.

Figura 4 – Um bloco de funções básico.



Fonte: Adaptado de Lewis (2008).

A norma IEC 61499 descreve o BFB como uma unidade funcional de *software* que possui sua própria estrutura de dados que pode ser manipulada por um ou mais algoritmos internos.

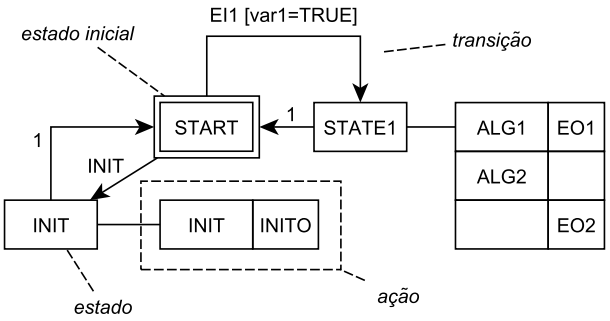
As variáveis de dados de entrada, saída e internas podem ser de qualquer tipo de dado já definido na norma IEC 61131-3, como BOOL, INT, REAL, etc. Os eventos são considerados sem tipo pela norma e não são chamados de variáveis. Logo, existem ambientes de execução que tratam os eventos como variáveis booleanas, outros que os tratam como números inteiros, o que permite a contagem de ocorrências de um determinado evento, e ainda, existem ambientes que o tratam como chamada de função.

Os algoritmos internos de um FB são executados a partir da ocorrência de um determinado evento, sob controle do ECC. A norma não define uma linguagem específica para a implementação desses algoritmos, portanto, assume-se que qualquer linguagem de programação pode ser utilizada. As ferramentas atuais tendem a utilizar as linguagens da norma IEC 61131-3 como LADDER e ST (*Structured Text*).

O ECC é uma máquina de estados que gerencia a execução do BFB. Os estados do ECC representam os possíveis estados do FB e as transições ocorrem quando um evento de entrada é acionado. A Figura 5 mostra um exemplo de um ECC.

Cada estado do ECC possui um conjunto de ações. Cada ação é formada por um algoritmo e um evento de saída do BFB. Ao atingir um determinado estado, as ações do mesmo

Figura 5 – ECC de exemplo.



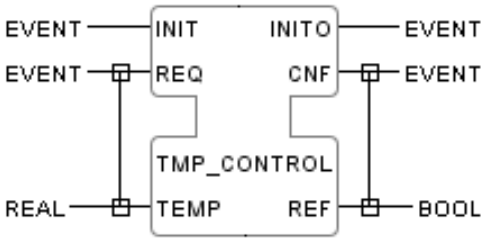
Fonte: Próprio Autor.

são executadas. Quando a ação contém um algoritmo e um evento, executa-se o algoritmo e, ao término deste, dispara-se o evento. Esse evento é um evento de saída do BFB. A ação pode conter apenas o algoritmo, nesse caso executa-se o algoritmo sem disparar um evento de saída. Também é possível conter apenas o evento de saída, nesse caso, apenas dispara-se o evento.

As transições são condicionadas a um evento de entrada do BFB e/ou a uma condição de guarda na forma *[condição]*. A evolução do ECC se dá pelos eventos de entrada. Entretanto, pode-se também utilizar condições de guarda, as quais acessam as variáveis do BFB para formar expressões lógicas.

Para exemplificar o funcionamento de um BFB, tem-se o BFB do tipo TMP_CONTROL, mostrado na Figura 6. Esse bloco executa o controle de temperatura, recebendo como entrada o valor de um sensor, e como saída, emite o valor lógico para ligar ou não um sistema de refrigeração.

Figura 6 – Um bloco de funções do tipo TMP_CONTROL.



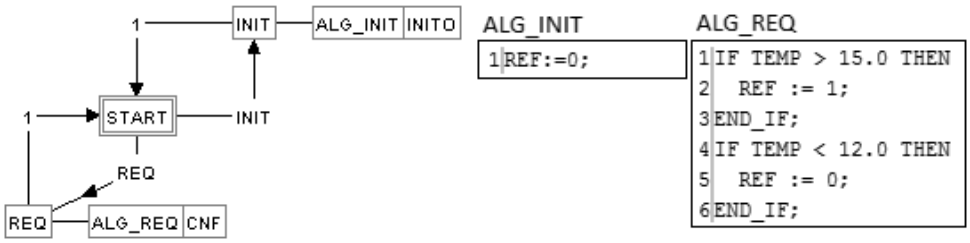
Fonte: Próprio Autor.

Nota-se a associação entre o evento REQ e a variável TEMP, essa associação indica que a variável TEMP será transferida para esse bloco quando o evento REQ ocorrer. Assim como a associação entre CNF e REF indica que REF será transferido quando CNF disparar.

O disparo do evento INIT causa a inicialização do bloco, a qual é feita por um algoritmo chamado ALG_INIT. A Figura 7 mostra o ECC do bloco de funções

TMP_CONTROL e seus algoritmos internos. O ECC inicia no seu estado inicial START, a ocorrência do evento INIT causa a mudança de estado para o estado INIT, nesse estado, o algoritmo ALG_INIT é disparado e, ao término deste, o evento INITO é disparado. De forma análoga, o evento REQ causa a mudança de estado para o estado REQ, executando o algoritmo ALG_REQ e o evento CNF.

Figura 7 – ECC do bloco TMP_CONTROL a esquerda e os algoritmos internos do bloco a direita.



Fonte: Próprio Autor.

O evento CNF indica que a temperatura foi processada e o valor em REF pode ser utilizado para acionar um sistema de refrigeração. Como mostra o algoritmo ALG_REQ, se TEMP for maior que 15.0, REF passa a ter o valor 1, indicando que o sistema de refrigeração deve ser ativado.

Nota-se o retorno automático do ECC para o estado START, através da transição 1 de INIT e REQ para START. Assim o ECC estará pronto para tratar novos eventos na entrada.

2.5.2 Bloco de Funções de Interface de Serviço

A norma IEC 61499 define o *Service Interface Function Block* (SIFB) como um bloco de funções que fornece um ou mais serviços para uma aplicação, ele faz o mapeamento desses serviços nos eventos e dados de entrada e saída.

A implementação de um SIFB requer conhecimento interno do dispositivo, o que pode envolver o sistema operacional e o hardware, sendo assim, os SIFBs são implementados pelos fabricantes do dispositivo. A IEC 61499 sugere que a interface de um SIFB contenha os eventos e variáveis de dados mostrados na Tabela 1.

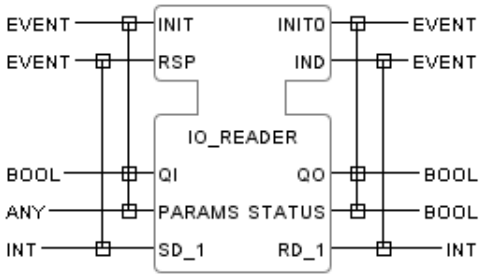
Como exemplo de SIFB, tem-se o bloco de funções do tipo IO_READER, mostrado na Figura 8. Esse bloco é padronizado pela IEC 61499, ele fornece acesso de leitura a entradas físicas do dispositivo. Esse bloco pode ser usado, por exemplo, para ler o valor de algum sensor conectado ao dispositivo.

Tabela 1 – Entradas e saídas padronizadas para SIFBs

Eventos de Entrada	
INIT	Inicializa o serviço fornecido pelo SIFB.
REQ	Solicita o serviço fornecido pelo SIFB. Numa operação de escrita, a aplicação precisa ativar REQ para que a operação seja efetuada.
RSP	Solicita o serviço fornecido pelo SIFB. Numa operação de leitura, a aplicação ativa RSP a fim de receber uma resposta.
Eventos de Saída	
INITO	Indica que a inicialização foi completada.
CNF	Esse evento confirma que uma operação de escrita foi concluída.
IND	Esse evento indica que uma operação de leitura foi concluída.
Dados de Entrada	
QI : BOOL	Se essa entrada é verdadeira durante a ocorrência do evento INIT, a inicialização do serviço é requisitada. Caso contrário, a finalização do serviço é requisitada.
PARAMS : ANY	Parâmetros para inicialização do SIFB. Numa operação de escrita, por exemplo, PARAMS pode conter o endereço de escrita.
SD_1, ..., SD_m : ANY	Dados de entrada que serão enviados ao serviço.
Dados de Saída	
QO : BOOL	Quando INITO dispara e QO é verdadeira, então o serviço foi inicializado com sucesso. Se falso então o serviço não foi inicializado com sucesso.
STATUS : ANY	Essa saída será de um tipo apropriado para expressar o estado do serviço sobre a ocorrência de um evento de saída.
RD_1, ..., RD_n : ANY	Essas saídas são valores recebidos do serviço.

Fonte: Próprio Autor

Figura 8 – Definição do SIFB IO_READER.

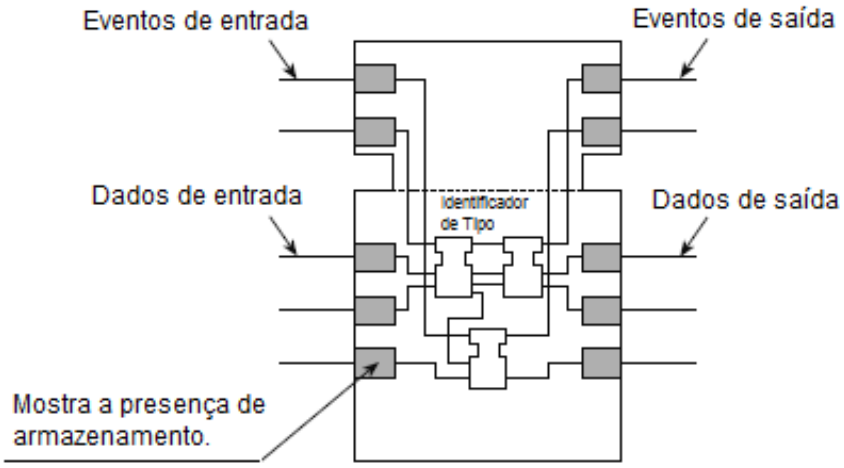


Fonte: Próprio Autor.

2.5.3 Bloco de Funções Composto

O *Composite Function Block* (CFB) ou bloco de funções composto é um tipo de bloco de funções que pode agrupar uma rede de blocos de funções. Normalmente utilizado para encapsular uma solução para um determinado problema. A Figura 9 mostra o modelo de um CFB.

Figura 9 – Um *Composite function block*.



Fonte: Adaptado de (LEWIS, 2008)

A interface externa de um CFB é idêntica aos demais blocos de funções, possuindo eventos e dados de entrada e saída. Porém internamente, o bloco contém uma rede de outros blocos de funções. Ele também possui capacidade de armazenamento de suas variáveis de entrada e saída. Os eventos do CFB são conectados a eventos dos blocos internos, assim como as variáveis são conectadas a variáveis desses blocos.

2.5.4 Rede de Blocos de Função

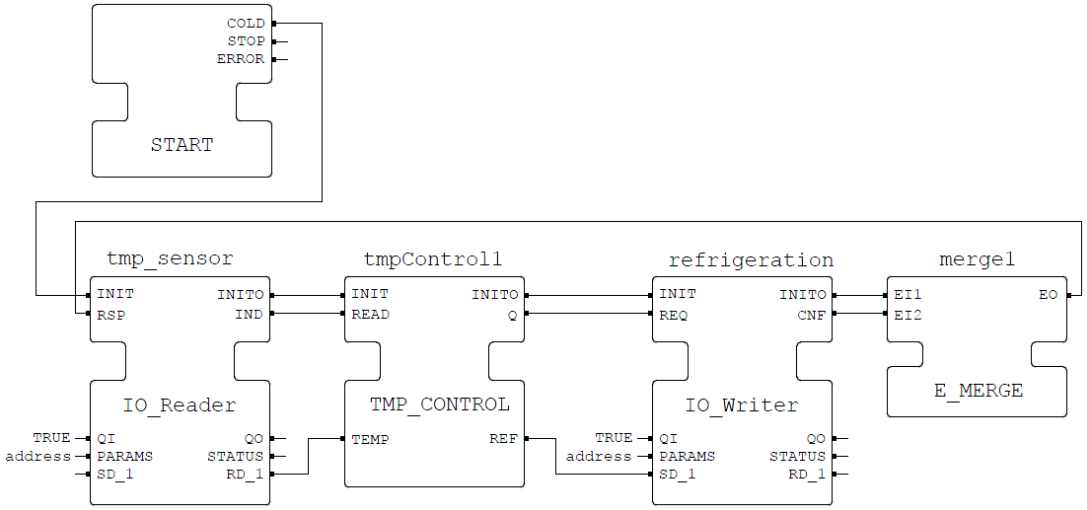
Dá-se o nome de rede de FBs (FBN) ao agrupamento de FBs através da interconexão de seus eventos e variáveis.

As variáveis de dados transportam informações de um FB para outro. Os eventos dos FBs não podem ser conectados as variáveis de dados do mesmo ou de outro FB.

A Figura 10 apresenta um exemplo de uma FBN. Trata-se de uma aplicação para controle de temperatura. O FB START é um bloco padrão da IEC 61499 e é responsável por disparar o evento inicial da aplicação. Como o fluxo de execução é controlado por eventos, faz-se necessário esse bloco para disparar o primeiro evento.

Esse primeiro disparo faz-se com que todos os blocos sejam inicializados em cascata. Após a inicialização do último bloco, o funcionamento da aplicação começa através do disparo do evento *tmp_sensor.RSP*. A aplicação passa a executar um loop infinito de leitura (*tmp_sensor*), processamento (*TMP_CONTROL*) e escrita (*refrigeration*).

Figura 10 – Um exemplo de uma FBN para controle de temperatura.



Fonte: Próprio Autor

2.6 MODELO DE GERENCIAMENTO

A IEC 61499 define uma forma especial de aplicação chamada “aplicação de gerenciamento”, a qual é responsável por criar redes de blocos de funções nos Recursos. A aplicação de gerenciamento possui maiores privilégios do que uma aplicação comum, ela pode construir outras aplicações nos Recursos do dispositivo em que a mesma se encontra. Geralmente, ela interage com agentes externos como ferramentas de configuração que permitem ao usuário editar um sistema aderente a norma IEC 61499.

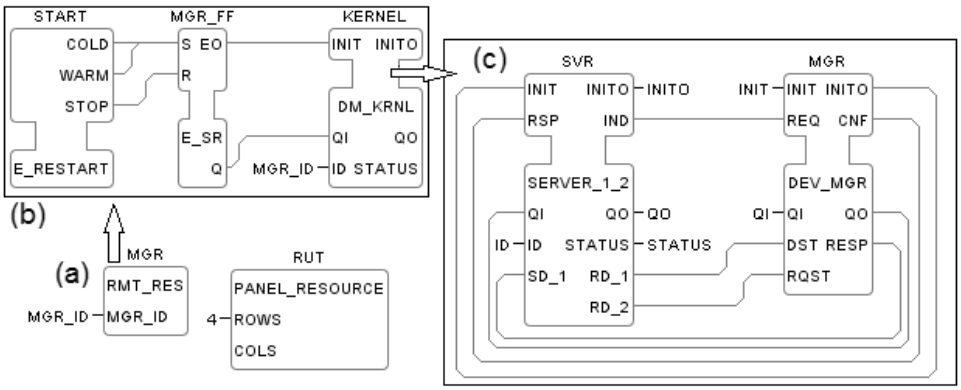
A aplicação de gerenciamento deve ser capaz de carregar redes de blocos de funções para diferentes aplicações sem perturbar a execução das demais aplicações em execução.

Geralmente a aplicação de gerenciamento está presente dentro do dispositivo numa forma não volátil. Ela já existe originalmente no dispositivo. Assim, o dispositivo é sempre

capaz de carregar aplicações quando ligado.

A Figura 11 mostra a aplicação de gerenciamento definida pelo perfil de conformidade da IEC 61499 (IEC 61499-4, 2013).

Figura 11 – Um dispositivo com aplicação de gerenciamento no Recurso MGR.



Fonte: Próprio Autor.

A Figura 11a mostra a visualização do dispositivo, o qual contém dois Recursos, sendo que o Recurso MGR é exclusivo para a aplicação de gerenciamento.

O bloco KERNEL, na Figura 11b, é um bloco de função composto, sendo que a Figura 11c mostra a rede de blocos de funções que o mesmo encapsula.

O tipo de bloco SERVER_1_2 é um FB capaz de criar um servidor TCP/IP, através do qual a aplicação recebe os comandos de reconfiguração. Os comandos recebidos são transferidos para o bloco MGR, do tipo DEV_MGR, o qual tem a capacidade de alterar aplicações em qualquer Recurso do dispositivo.

A aplicação de gerenciamento pode executar um conjunto de comandos, os quais tem a capacidade de criar e excluir blocos de funções, assim como as conexões entre eles. Os blocos de funções também podem ser inicializados, reiniciados ou parados através destes comandos (DAI et al., 2015).

2.7 AMBIENTES DE EXECUÇÃO ADERENTES A IEC 61499

Nessa seção são apresentados alguns ambientes de execução para a norma IEC 61499 e seus principais detalhes de implementação.

A maioria das soluções IEC 61499 utilizam algum tipo de ambiente de execução como o FORTE (Eclipse 4DIAC, 2019), o FBRT (HOLOBLOC Inc., 2018b) e o ICARU-FB (PINTO et al., 2016). Esses ambientes, entretanto, são implementados cada um com um modelo de execução diferente.

Umas das principais dificuldades da implementação de ambientes de execução é com relação as diferentes interpretações da norma IEC 61499. Mesmo que as implementações dos ambientes de execução estejam de acordo com a norma IEC 61499, é possível que eles apresentem comportamentos diferentes durante a execução (VYATKIN, 2009; DUBININ; VYATKIN, 2012; CENGIC; AKESSON, 2010).

Ferrarini e Veber (2004) apresentam e comparam algumas abordagens para a implementação de um ambiente de execução. A seguir destacam-se duas possíveis implementações:

Implementação I1: Cada bloco de funções é implementado como um objeto, enquanto que todos os eventos são manipulados por um único objeto. Um bloco de funções contém as variáveis internas como atributos e os algoritmos como métodos e o ECC como o método $run()$. Todos os ECCs também possuem um método $initialize()$. O objeto que manipula eventos contém um método chamado $setEvent(event)$ que sequencialmente chama o método $run()$ dos blocos de função conectados a $event$. A geração de eventos de saída é implementada no método $run()$ do bloco de funções como uma chamada direta de $setEvent(event)$ do objeto que manipula eventos.

Implementação I2: Cada bloco de funções é implementado como um objeto que estende a classe base $Thread(Runnable)$. Esse objeto contém os mesmos atributos e métodos descritos em I1 e uma instância da classe $EventBuffer$. O objeto $EventBuffer$ é instanciado na chamada de $initialize()$ e implementa a interface de eventos de entrada. O método $setEvent(event)$ da classe $EventBuffer$ adiciona $event$ em uma lista de eventos ocorridos e executa a função $NotifyAll()$. A conexão de eventos é implementada como em I1, mas $setEvent(event)$ chama sequencialmente o método $setEvent(event)$ do $eventBuffer$ correspondente. Devido a implementação de $EventBuffer$, essa operação termina em um intervalo de tempo pré-fixado, diferente do que acontece em I1.

Quando um ECC é "acordado" por $NotifyAll()$, ele lê sequencialmente a lista de eventos ocorridos até que uma transição possa ser apurada. Se nenhuma transição ocorrer, o ECC suspende sua execução.

As implementações I1 e I2 foram realizadas por Ferrarini e Veber (2004) em Java e comparadas entre si. I1 oferece vantagem com relação a velocidade de execução quando a rede de blocos de função é em série, mas apresenta comportamentos inesperados quando há um loop na conexão de eventos e quando mais de um evento precisa ser gerado ao mesmo tempo. A implementação I2 apresenta velocidade comparável a implementação I1 quando a rede é composta por blocos de função em paralelo e é a implementação que apresenta o comportamento mais esperado de acordo com a norma IEC 61499.

O FBRT (HOLOBLOC Inc., 2018b) foi o primeiro ambiente de execução implementado para a norma IEC 61499. É considerado uma implementação de referência e também utilizado para testar a norma, ele segue o modelo de implementação I1. Esse ambiente é implementado em Java (HALL; STARON; ZOITL, 2007).

No FBRT, um bloco de funções é implementado como um objeto. Os eventos de saídas que ativam outros blocos de função, são implementados como chamadas de função ao bloco de funções de destino, isso significa que a execução do bloco de funções atual é parada para que seja executada uma função de outro bloco de funções (YOONG; ROOP; SALCIC, 2009; FERRARINI; VEBER, 2004).

O ambiente de execução FORTE tem uma implementação semelhante à implementação I2. Segundo Strasser et al. (2008), o objetivo desse ambiente é ser uma solução aberta e que possa ser utilizada na indústria atendendo aos requisitos de portabilidade, configurabilidade e interoperabilidade.

O FORTE é implementado em C++. Os blocos de funções são implementados em classes, o disparo de um evento sobre um bloco de funções faz com que este seja colocado em uma fila do tipo *first-in-first-out*. Quando o bloco de funções que originou o evento terminar sua execução, o próximo bloco de funções na fila será executado.

O FBC, proposto em Yoong, Roop e Salcic (2009), é um compilador que utiliza como entrada um arquivo *xml* e gera código em linguagem C para ser compilado e executado sem a utilização de um ambiente de execução. Como mostrado por Yoong, Roop e Salcic (2009), essa abordagem traz vantagens com relação ao desempenho, pois elimina a necessidade de um ambiente de execução.

O FBC implementa os blocos de função como estruturas em linguagem C e alguns métodos de execução. A estrutura de dados guarda o estado do ECC, eventos de entrada e saída e os dados de entrada e saída. As conexões entre os blocos de função são implementadas utilizando memória compartilhada (KUO et al., 2010).

O ICARU-FB é composto por uma máquina virtual (ICARU-VM) e uma ferramenta de atualização dinâmica (Dynamic Update Tool - DUT). A ICARU-VM possui o seu próprio conjunto de instruções, o qual foi projetado para executar os blocos de funções em plataformas com poucos recursos computacionais. Os blocos de funções são compilados em blocos de instruções e enviados a máquina pela DUT.

Além dos ambientes citados anteriormente, também existem ISaGRAf (Rockwell Automation, 2019) e nxtRT61499F (Schneider Electric, 2015) que são soluções fechadas e não puderam ser avaliadas como os demais ambientes. Entretanto, foi possível verificar que ambos os ambientes necessitam de um sistema operacional para o seu funcionamento.

2.8 RECONFIGURAÇÃO DINÂMICA

O modelo de Gerenciamento da norma IEC 61499, permitiu a implementação da reconfiguração dinâmica. A reconfiguração dinâmica é a habilidade do dispositivo de alterar as aplicações em tempo de execução. Isso também se estende a alterações de hardware, permitindo que os dispositivos sejam acrescentados ou removidos de um determinado sistema, sem que o mesmo seja desligado ou reiniciado.

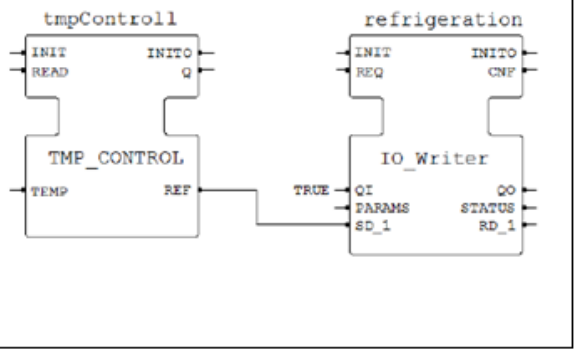
Sistemas reconfiguráveis dinamicamente podem ser modificados sem a necessidade de recompilar todo o *software* ou reiniciar o sistema. As características dos blocos de funções os tornam adequados a estes tipos de aplicações, pois são executados de forma atômica.

Há vários trabalhos na área de reconfiguração dinâmica envolvendo a IEC 61499. Eles serão apresentados no Capítulo 4.

A IEC 61499 não define uma lista de comandos para a reconfiguração dinâmica dos dispositivos, mas um perfil de conformidade (*compliance profile*) foi criado para uma tentativa de demonstrar a viabilidade de um conjunto conciso de comandos necessários (VYATKIN, 2012). A Figura 12 mostra um exemplo de comandos de gerenciamento enviados ao dispositivo para criar uma rede de blocos de função. Os comandos são enviados em formato *xml* ao dispositivo, o qual executa as ações solicitadas.

Figura 12 – Um exemplo de comandos de gerenciamento criando uma rede de blocos de função.

#	Comando
1	<Request ID="3" Action="Create"> <FB Name="tmpControl1" Type="TMP_CONTROL"/> </Request>
2	<Request ID="4" Action="CREATE"> <FB Name="refrigeration" Type="IO_Writer"/> </Request>
3	<Request ID="5" Action="CREATE"> <Connection Source="tmpControl1.REF" Destination="refrigeration.SD_1"/> </Request>
4	<Request ID="6" Action="WRITE"> <Connection Source="TRUE" Destination="refrigeration.QI"/> </Request>



Fonte: Adaptado de (VYATKIN, 2012).

O comando 1 cria uma instância do bloco de funções TMP_CONTROL com o nome *tmpControl1*. O comando 2 cria uma instância do bloco de funções IO_Writer com o nome *refrigeration*. O comando 3 cria uma conexão de dados. E o comando 4 cria uma conexão com uma constante TRUE.

Cada comando de gerenciamento é seguido de uma resposta do dispositivo, informando o resultado da execução do comando.

2.9 CONSIDERAÇÕES DO CAPÍTULO

A norma IEC 61499 padroniza o desenvolvimento de sistemas distribuídos. Essa norma está em evolução e ainda possui alguns problemas com relação a ambiguidades na sua semântica. Ainda não existe um modelo formal capaz de definir o comportamento preciso de todos os elementos da IEC 61499. Consequentemente, existem várias implementações de ambientes de execução incompatíveis entre si. Isso é contrário a proposta da norma, a qual tenta garantir a portabilidade das aplicações em diferentes ambientes de execução.

Existem trabalhos com o objetivo de formalizar os modelos apresentados pela norma IEC 61499, os quais serão apresentados no Capítulo 4. Entretanto esses formalismos não são completos, sendo que a maioria foca somente no modelo de FB da norma, deixando a FBN, por exemplo, sem definição formal. Por este motivo, o método de RD proposto nesta tese se baseia em sistemas a eventos discretos (SEDs), que possuem uma base sólida e podem ser totalmente formalizados por autômatos. O Capítulo 3 apresentará o conceito de autômato nos quais a Teoria de Controle Supervisório (TCS), apresentada nesta tese, é baseada. Posteriormente, no Capítulo 6 será apresentada a aplicação do método proposto em sistemas aderentes a IEC 61499.

3 CONTROLE SUPERVISÓRIO DE SISTEMAS A EVENTOS DISCRETOS

Este capítulo visa dar suporte aos conceitos de Autômatos Finitos Determinísticos (AFD) (HOPCROFT et al., 2000), Sistemas a Eventos Discretos (SEDs) e a Teoria do Controle Supervisório (TCS) (RAMADGE; WONHAM, 1982).

Um SED é um sistema dinâmico que evolui de acordo com a ocorrência de eventos discretos em intervalos irregulares e, possivelmente, desconhecidos. Por exemplo, um evento pode indicar o fim de operação de uma máquina numa linha de manufatura ou a chegada de peças numa determinada posição através de uma esteira.

A TCS, inicialmente proposta por Ramadge e Wonham (1982), permite a síntese automática da lógica de controle em malha fechada minimamente restritiva para Sistemas a Eventos Discretos (SEDs), impedindo somente sequências de eventos em desacordo com as especificações impostas pelo projetista. A TCS não é obrigatória para a reconfiguração dinâmica segura (RDS) proposta nesta tese no Capítulo 5, somente o conceito de eventos controláveis é utilizado. Entretanto, a lógica de controle gerada através da TCS é a menos restritiva, logo, a RDS também obtém um resultado menos restritivo, permitindo que o SMR realize mais tarefas durante a RD.

Este capítulo é organizado da seguinte forma, a Seção 3.1 apresenta os sistemas a eventos discretos. As Seções 3.2 e 3.3 os conceitos de linguagens e autômatos respectivamente. A Seção 3.4 ilustra um exemplo de um SED. A TCS é apresentada na Seção 3.5. Por fim, a Seção 3.6 apresenta as considerações desse Capítulo

3.1 SISTEMAS A EVENTOS DISCRETOS

Quando o espaço de estados de um sistema é naturalmente descrito por um conjunto discreto como $\{0, 1, 2, ...\}$, e as transições entre estados são observáveis somente em pontos discretos no tempo, associam-se essas transições a eventos e chama-se esse sistema de Sistema a Eventos Discretos (SED) (CASSANDRAS; LAFORTUNE, 2008). A ocorrência desses eventos é considerada com tempo nulo (ocorrem instantaneamente).

Nos SMR, os eventos representam sinais elétricos como, por exemplo, iniciar operação de uma máquina e o sinal de um sensor. Num SMR, o dispositivo que executa a LC faz o interfaceamento dos sinais elétricos com os eventos, os quais são associados as entradas e saídas do dispositivo.

Um SED pode ser modelado por um autômato, o qual possui estados e transições. O formalismo de autômatos é empregado novamente no Capítulo 5 desta tese.

3.2 LINGUAGENS

Uma linguagem é um conjunto de palavras formadas através dos símbolos de um alfabeto. Por exemplo, a linguagem portuguesa é um conjunto de palavras formadas sobre o alfabeto latino. No contexto de SEDs, o alfabeto é o conjunto de eventos enquanto um símbolo é um dos eventos. Uma palavra, que é um sequência de símbolos, é dito uma sequência de eventos. Uma vez que eventos são associados aos sinais elétricos do dispositivo, uma sequência de eventos representa uma sequência de ações no sistema.

Uma linguagem é denotada por L. Uma palavra de uma linguagem é denotada por w. O alfabeto, que é um conjunto de símbolos finito, é denotado por Σ. A seguir tem-se alguns exemplos de alfabetos:

1. $\Sigma = \{0, 1\}$, o alfabeto binário;
2. $\Sigma = \{a, b, c, d, ..., y, z\}$, o alfabeto das letras minúsculas.

Define-se uma palavra (ou cadeia) w como sendo um sequência construída a partir dos símbolos de um alfabeto Σ. Por exemplo, a palavra 0111 é construída a partir do alfabeto $\Sigma = \{0, 1\}$. O conjunto de todas as palavras possíveis formadas a partir do alfabeto Σ é denotado por Σ^*.

Denota-se $|w|$ como sendo a quantidade de símbolos ou tamanho da palavra w. Por exemplo, $|0111| = 4$. Quando $|w| = 0$ tem-se uma palavra vazia, formada por nenhum símbolo, a qual se denota por ε.

Um conjunto de palavras selecionadas a partir de Σ^*, onde Σ é um alfabeto, é chamado de linguagem e denotado por L. Diz-se que L é uma linguagem sobre Σ quando $L \subseteq \Sigma^*$. Não necessariamente L terá todas as palavras de Σ^*, por exemplo, o Português é uma linguagem L sobre o alfabeto Σ de letras, mas nem todas as palavras de Σ^* fazem parte da linguagem. São exemplos de linguagens sobre $\Sigma = \{a, b\}$:

1. $L = \{ab, aab, aba\}$, uma linguagem com três palavras;
2. $L = \{w | w$ começa com b e $|w| = 4\}$, uma linguagem finita com palavras como $baaa$ e $bbba$;
3. $L = \{w | w$ contém um número par de a's.$\}$;

Nesse trabalho, será utilizada a classe de linguagens regulares, que é a classe na qual a TCS se baseia. Uma linguagem é dita regular quando pode ser reconhecida por um autômato.

3.3 AUTÔMATO FINITO DETERMINÍSTICO

O comportamento de SED pode ser modelado por autômatos. Um Autômato, também chamado de máquina de estados, é um formalismo capaz de reconhecer alguma linguagem regular.

Nesse trabalho será apresentado o Autômato Finito Determinístico (AFD). Para a classe de linguagens regulares, existem ainda os autômatos não determinísticos e os com movimento vazio. Porém, todos têm o mesmo poder computacional e podem ser transformados em um AFD. A definição de um AFD é como segue:

Definição 3.1. *Um AFD é definido como uma 5-tupla (HOPCROFT et al., 2000):*

$$G = (Q, \Sigma, \delta, q_0, F)$$

na qual:

Q *é um conjunto de estados;*

Σ *é um conjunto de símbolos, chamado alfabeto;*

δ *é uma função de transição* $\delta : (Q \times \Sigma) \rightarrow Q$*;*

q_0 *é o estado inicial;*

F *é um conjunto de estados finais.*

Considere o autômato $G = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$, cuja função de transição é definida pela Tabela 2. Outra forma de representar δ é indicando o resultado para cada elemento de $Q \times \Sigma$, se δ estiver definido para tal:

$$\delta(q_0, a) = q_1$$
$$\delta(q_1, a) = q_1$$
$$\delta(q_1, b) = q_1$$

Tabela 2 – Função de transição δ do Autômato G.

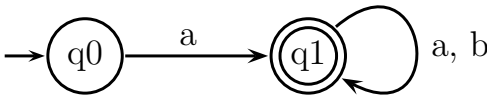
δ	a	b
q_0	q_1	$-$
q_1	q_1	q_1

Fonte: Próprio Autor

A representação visual do mesmo autômato G é mostrada na Figura 13. Esse autômato reconhece a linguagem cujas palavras começam com a. O processamento da palavra pelo autômato se dá aplicando-se a função de transição sobre cada símbolo da palavra a ser

reconhecida. Ao terminar de processar toda a palavra, se o autômato parou num estado $q \in F$, então a palavra pertence à linguagem reconhecida pelo autômato. No contexto de SEDs, o processamento do autômato nunca termina, pois tem-se um sistema em malha fechada que continuamente executa tarefas. Logo, a passagem por um estado $q \in F$ significa que o sistema completou alguma tarefa.

Figura 13 – Representação gráfica do autômato G.



Fonte: Próprio Autor

Sendo $w = abba$ a palavra a ser reconhecida por G, inicia-se aplicando $\delta(q_0, a)$ que resulta em q_1. A partir daí segue:

$$\delta(q_0, a) = q_1 \tag{3.1}$$
$$\delta(q_1, b) = q_1 \tag{3.2}$$
$$\delta(q_1, b) = q_1 \tag{3.3}$$
$$\delta(q_1, a) = q_1 \tag{3.4}$$

De forma similar, pode-se observar as mesmas transições na Figura 13. Como termina-se no estado $q_1 \in F$, então a palavra w faz parte da linguagem denotada por G. A aplicação sequencial de δ sobre a palavra w pode ser denotada de forma resumida por $\hat{\delta}(q_0, w) = q_1$, o que é conhecido como função de transição estendida ($\hat{\delta}$). Denota-se ainda a linguagem reconhecida por G como $L(G) = \{w | \hat{\delta}(q_0, w) \in F\}$. Ou seja, $L(G)$ é a linguagem com todas as palavras reconhecidas por G.

Na área de SEDs, algumas obervações devem ser feitas sobre o uso dos autômatos. Como o sistema opera em malha fechada, diz-se que os estados finais são na verdade estados marcados. O conceito de palavra aceita pelo autômato passa a significar tarefa completada pelo SED. Esse não irá parar neste estado, ele continuará evoluindo pelos estados e cada passagem por um estado marcado significa uma tarefa completada. Há ainda o conceito de linguagem gerada e marcada. A linguagem gerada em SEDs é qualquer sequência de eventos ou palavra que leve a um estado qualquer, não apenas ao marcados. A linguagem marcada são as sequências de eventos que levam aos estados marcados.

3.4 EXEMPLO DE UM SED

O comportamento de um SED pode ser modelado por um AFD. Considere o SED da Figura 14. Nesse sistema há 3 máquinas distintas m_i com $i \in \{1, 2, 3\}$ e dois buffers b_j com

$j \in \{1, 2\}$. Cada máquina possui dois eventos sendo s_i para dar inicio de operação da mesma e f_i para indicar o término de operação.

Figura 14 – Planta sequencial.

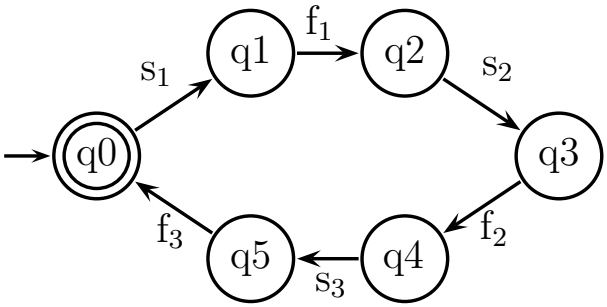
Fonte: Adaptado de (TEIXEIRA; CURY; QUEIROZ, 2011)

Um controle simples desse sistema poderia ser executado da seguinte forma:

1. O controlador dispara s_1. Então m_1 coleta uma peça da entrada, executa um processo, deposita-a em b_1 e dispara f_1;
2. Ao detectar f_1, o controlador pode disparar s_2 para iniciar a operação de m_2, a qual coleta a peça de b_1 e efetua um processo disparando f_2 ao depositar uma peça em b_2;
3. Ao detectar f_2, o controlador pode disparar s_3 para iniciar a operação de m_3, a qual coleta a peça de b_2 e efetua um processo e disparando f_3 depositando a peça no final da linha;
4. Detectando f_3, o controlador pode iniciar um novo processo disparando s_1 e voltando ao passo 1.

Esse comportamento em malha fechada do SED é imposta ao sistema pelo controlador e pode ser representado pelo autômato na Figura 15. Observa-se que, apesar do autômato ter um estado final, não significa que se deve esperar uma parada do sistema neste estado. No contexto de SEDs para automação e controle, o estado final poderia ser melhor entendido como o término de alguma tarefa do sistema, o qual provavelmente opera em malha fechada, sem nunca parar. Assim, uma possível sequência de eventos seria $s_1 f_1 s_2 f_2 s_3 f_3 s_1 f_1 \ldots$

Figura 15 – Autômato para controle da planta na Figura 14.



Fonte: Próprio Autor

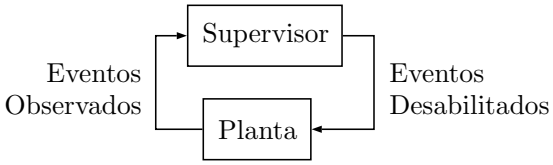
Nota-se que essa LC é restritiva ao ponto de só permitir uma nova entrada de peça na linha quando m_3 termina a sua operação. Apenas uma peça é produzida por vez e somente

uma máquina trabalha a cada ciclo, enquanto as demais permanecem ociosas. Uma lógica de controle melhor poderia ser obtida, como por exemplo, disparar s_1 logo após o disparo de s_2, possibilitando a operação em paralelo de m_1 e m_2. Mas tratar todas as possíveis operações, de forma que o sistema execute o maior número de tarefas em menos tempo, pode ser inviável para construção manual. Através da TCS, pode-se obter a melhor lógica de controle do sistema de forma sistemática.

3.5 CONTROLE SUPERVISÓRIO DE SEDS

Na TCS, tem-se o conceito de planta e supervisor. Ambos são modelados por autômatos. O modelo da planta representa a mesma sem nenhum tipo de controle imposto, assim, no exemplo ilustrado na Figura 14 o modelo de planta pode inicialmente disparar o evento s_3, que não é permitido. Para que a planta seja controlada, o supervisor observa os eventos da mesma e desabilita eventos a fim de que a planta se comporte de acordo com as especificações. Esse controle de malha fechada é mostrado na Figura 16.

Figura 16 – Controle em malha fechada.



Fonte: (CASSANDRAS; LAFORTUNE, 2008)

A obtenção do supervisor é realizada através de uma sequência de passos sistemática, cujos cálculos, são atualmente auxiliados por ferramentas como Nadzoru (PINHEIRO et al., 2015), TCT (FENG; WONHAM, 2006) e IDES(RUDIE, 2006). Os passos são:

1. Obtenção de um modelo para a planta;
2. Modelagem das especificações, que são as regras de controle;
3. Cálculo da máxima linguagem controlável.

A TCS divide o alfabeto em eventos controláveis (Σ_c) e não controláveis (Σ_u). Assim, o conjunto de eventos do SED é a união disjunta $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$. Os eventos controláveis são aqueles que podem ser impedidos pelo supervisor. Por exemplo, o evento que inicia a operação de uma máquina pode ser controlado, ou seja, pode ser habilitado ou desabilitado pelo supervisor. Um evento não controlável é, por exemplo, quando uma máquina indica que terminou uma operação, este não tem como ser desabilitado.

Normalmente, os eventos controláveis podem ser associados aos sinais de saída de um controlador (como um CLP por exemplo) e os não controláveis aos sinais de entrada. Entretanto, podem haver eventos que não são associados a tais sinais. Um exemplo é o disparo e o fim de contagem de um temporizador, o qual é implementado internamente no controlador.

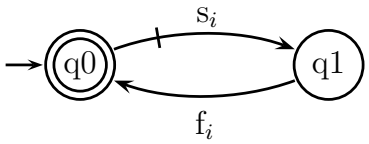
Obtenção de um Modelo de Planta

A planta é modelada por um AFD denotado por G, o qual representa o comportamento do sistema sem nenhuma especificação de controle. Esse modelo é obtido a partir de modelos (AFDs) menores que são os modelos dos elementos que compoem a planta, como máquinas, robôs, sensores, etc.

O modelo de um elemento de planta é representado por um AFD, o qual também pode ser chamado de subsistema. O AFD representa a sequência de eventos que se pode observar do elemento. O Exemplo 3.1 mostra o modelo para uma máquina.

Exemplo 3.1. *A operação da máquina m_1, ilustrada na Figura 14, consiste na sequência $s_i \rightarrow f_i$, que significam inicio de operação e término de operação da máquina m_1. Como s_i é disparado por um controlador, diz-se que este evento é controlável. Na representação visual, como na Figura 17, os eventos controláveis são representado com um traço sobre a transição. O evento f_i é não controlável, pois não pode ser impedido por um controlador, na verdade, ele é lido.*

Figura 17 – Autômato para a máquina m_i. 14.



Fonte: Próprio Autor

Tendo um modelo G_i para cada elemento i da planta, faz-se a operação de composição síncrona, denotada por $||$, entre todos os modelos G_i. Então, obtém-se um modelo G da planta da seguinte forma:

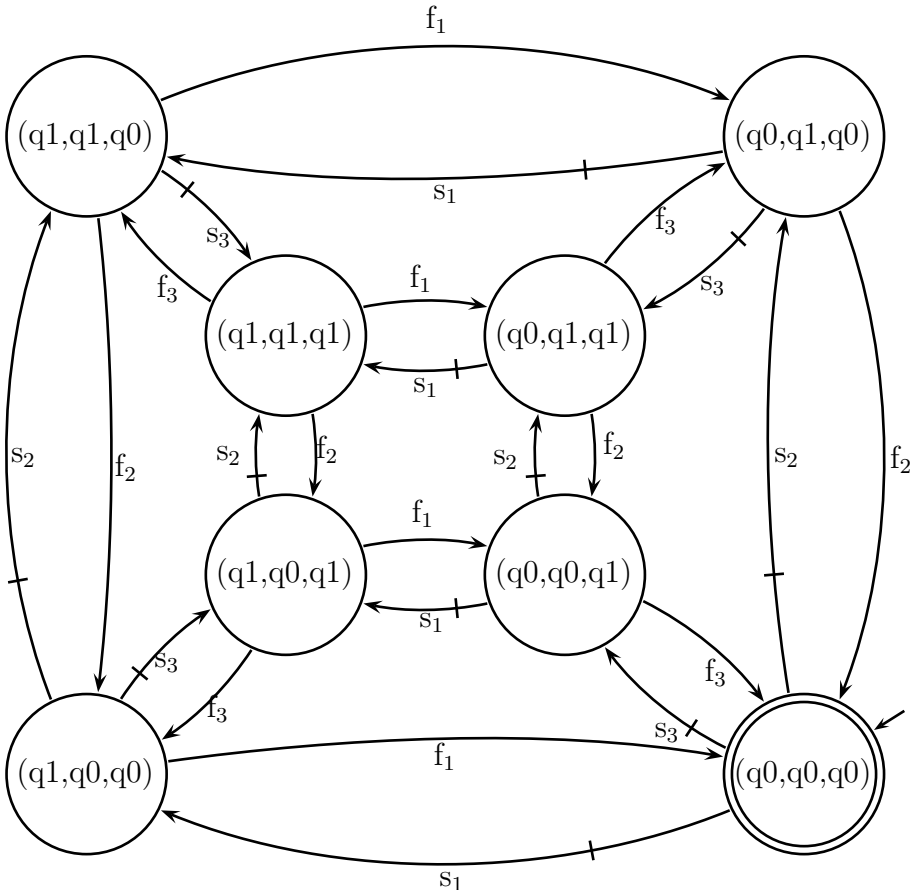
$$G = \Big|\Big|_{i=1}^{n} (G_i)$$

O modelo G representa o comportamento da planta sem nenhum controle. Ou seja, qualquer evento controlável pode ser executado pela planta a qualquer momento, exceto se o modelo do elemento não permite. Observa-se que a operação síncrona entre os modelos G_i, gera um autômato que representa o comportamento concorrente de todos os subsistemas G_i.

Exemplo 3.2. *A Figura 18 mostra o modelo G obtido para $G = G_1||G_2||G_3$. Observa-se que logo no estado inicial, é possível que a planta dispare o evento s_3, o que inicia a operação da máquina m_3.*

O que sabe-se que, apesar desse comportamento ser fisicamente possível, ele não é desejado, pois permite o início da máquina m_3 sem que haja peça no buffer de entrada.

Figura 18 – Modelo de planta G que representa o comportamento fisicamente possível.



Fonte: Próprio Autor

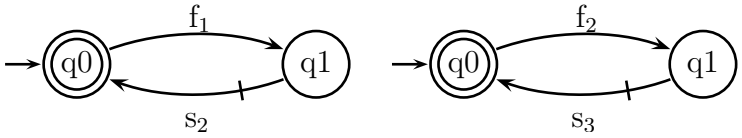
Modelagem das Especificações

O passo seguinte consiste em modelar as especificações de controle do sistema. Cada especificação E_j é modelada através de um AFD. Utilizando o mesmo processo de obtenção da planta, faz-se a composição síncrona de todas as especificações, obtendo-se um modelo denotado por E.

As especificações são modeladas observando-se somente a restrições desejadas. Deve-se apenas restringir o comportamento não desejado, permitindo que todos os demais comportamentos possam ocorrer. A seguir, tem-se exemplos de especificação.

Exemplo 3.3. *As restrições necessárias para o correto funcionamento da planta na Figura 14, consistem (a) impedir m_2 de iniciar sem peças em b_1, o que corresponde ao não underflow do buffer e (b) impedir m_3 de inciar sem peças em b_2. O término de operação de m_1, denotado pelo evento f_1, indica que uma peça foi colocada em b_1. Assim, s_2 pode ser disparado. O mesmo vale para m_2 e b_2. As duas especificações são como mostrado na Figura 19.*

Figura 19 – Especificações para a planta da Figura 14.



(a) E_1 (b) E_2

Fazendo a composição síncrona de todas as especificações, obtém-se E:

$$E = \Big|\Big|_{j=1}^{n} (E_i)$$

Obtenção do Supervisor

Modela-se o comportamento desejado para a planta G fazendo a composição $H = G||E$. Entretanto, na prática, esse comportamento pode contemplar uma lógica que proíba a ocorrência de eventos não controláveis na planta, o que não é passível de ser realizado. A linguagem de H é denotada por $K = L(H)$.

O último passo é a obtenção do supervisor, que consiste em calcular a máxima sublinguagem de K que é controlável em relação a planta. O supervisor é a LC sem bloqueios e formado apenas por estados que representam comportamentos corretos da planta.

O sistema controlado por um supervisor é denotado por S/G, que significa "S controlando G". O comportamento em malha fechada do sistema é $L(S/G) = SupC(K, L(G))$.

Um supervisor é um mapa $S : L(G) \rightarrow 2^{\Sigma}$ que especifica, para cada possível cadeia de eventos $s \in L(G)$, os eventos habilitados no estado corrente de G. Por exemplo, na planta ilustrada na Figura 18, O único evento habilitado no estado inicial será o evento s_1.

O supervisor S pode ser representado por um autômato. Esse pode ser utilizado como base para criar ou gerar código para um controlador da planta.

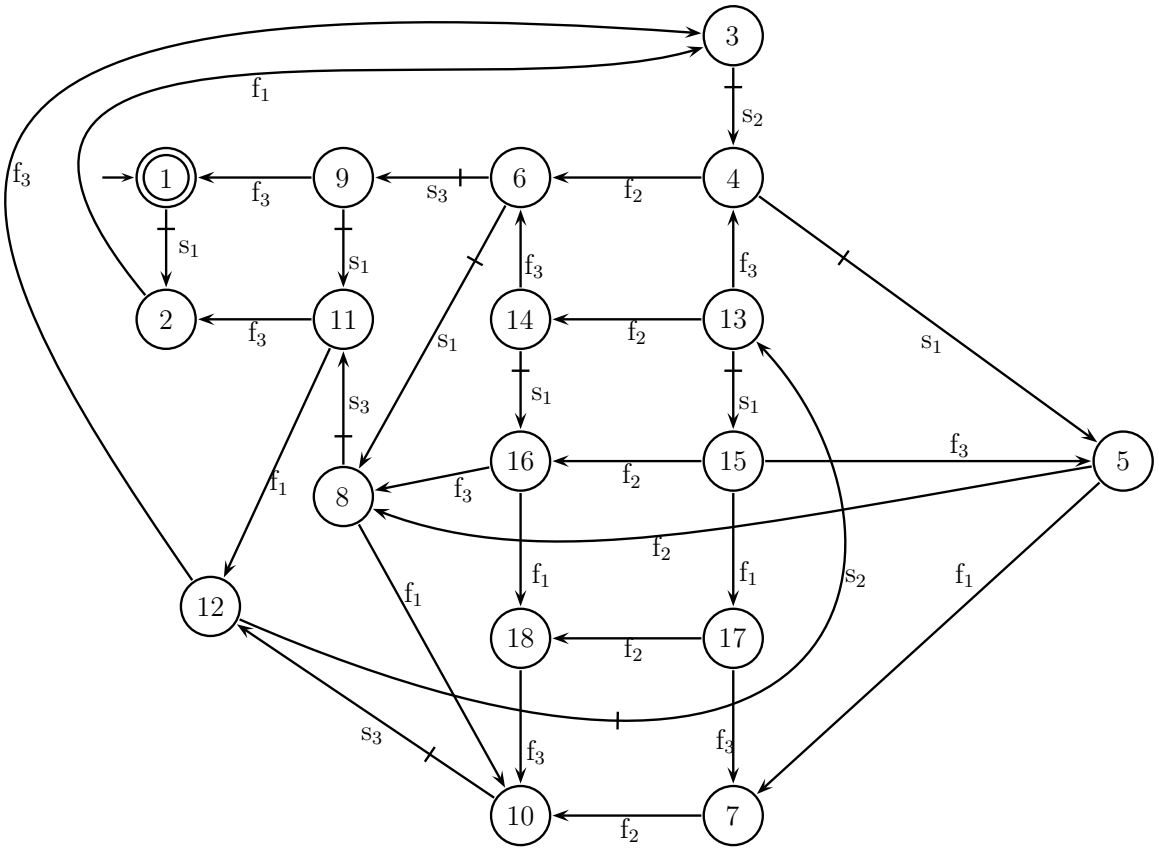
Exemplo 3.4. *Fazendo a operação $L(S/G) = SupC(K, L(G))$ onde:*

- $G = G_1||G_2||G_3$*: O modelo da planta sem supervisão;*
- $E = E_1||E_2||E_3$*: A composição das especificações;*

- $H = G||E$: *O modelo do comportamento da planta que pode conter bloqueios;*
- $K = L(H)$: *A linguagem do comportamento da planta.*

Obtém-se S que pode ser representado pelo autômato na Figura 20.

Figura 20 – Supervisor para a planta da Figura 14.



Fonte: Próprio Autor

Observa-se que a lógica de controle gerada é muito mais elaborada do que aquela feita manualmente na Figura 15. No estado 4 por exemplo, há a possibilidade de disparar s_1, pois s_2 já foi disparado fazendo com que m_2 retire a peça de b_1. Nota-se ainda, que quando b_1 (estados 3, 7, 10, 12, 17, 18), a máquina m_1 fica impedida de iniciar tarefa, pois em nenhum desses estados é permitido executar s_1.

3.6 CONSIDERAÇÕES DO CAPÍTULO

Esse capítulo apresentou os principais conceitos de SEDs, linguagens, autômatos e da TCS. SEDs são abordados nessa tese pois são a base da proposta apresentada no Capítulo 5. O formalismo utilizado para modelar os SEDs é o AFD, que será amplamente utilizado nesta tese. A TCS fornece um método sistemático que permite encontrar uma lógica de controle ótima

para um SED. Isso se torna útil para o método da reconfiguração dinâmica segura proposto neste trabalho, pois permite também uma RD menos restritiva.

Os sistemas de automação industrial, como o SMR, podem ser modelados por SEDs. Mesmo sistemas de natureza contínua podem ser abstraídos em SEDs. Por exemplo, pode-se atribuir uma valor limite a um sensor de temperatura e disparar um evento quanto este atingir determinado valor.

Os trabalhos de RD na área de SEDs apresentam um formalismo maior quando comparado aos trabalhos de RD da área da IEC 61499. Isso se deve ao fato dos SEDs utilizarem formalismos bem definidos como os autômatos.

4 TRABALHOS RELACIONADOS

Desde a primeira publicação da norma IEC 61499, uma série de trabalhos foram desenvolvidos a seu respeito. Nessa seção, serão apresentados aqueles que apresentam algum método para modelagem dos elementos da norma ou alguma técnica de reconfiguração dinâmica. Além disso, serão também apresentados trabalhos de reconfiguração dinâmica envolvendo sistemas a eventos discretos (SEDs).

A possibilidade de reconfigurar dinamicamente um sistema industrial é um tópico de pesquisa e desenvolvimento há muitos anos (STRASSER et al., 2014). Existem vários trabalhos abordando a reconfiguração dinâmica em sistemas aderentes a norma IEC 61499 (OLSEN et al., 2005; ZOITL; SUNDER; TERZIC, 2006; ALMEIDA; LUNTZ; TILBURY, 2007; Guellouz et al., 2016). A maior parte deles foca em como reduzir o impacto da reconfiguração no processo controlado e, também, em como manter a consistência da aplicação de controle durante e após a reconfiguração (ZOITL; SUNDER; TERZIC, 2006). O gerenciamento do processo de reconfiguração é chamado gerenciamento de transição.

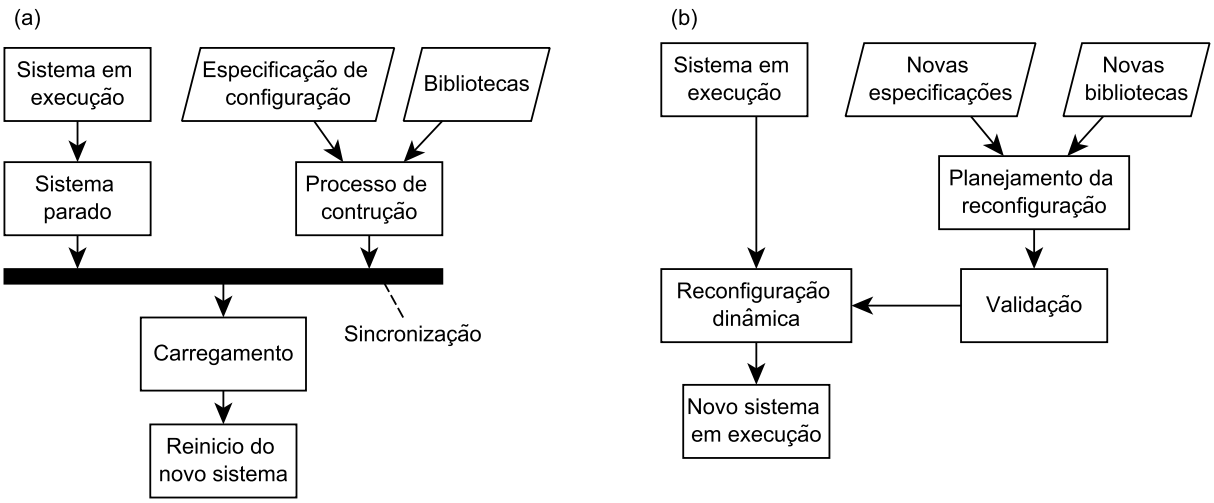
Segundo Zoitl, Sunder e Terzic (2006), a reconfiguração dinâmica (RD) é possível somente tendo conhecimentos específicos do SMR. Para suportar esse recurso, é necessário um serviço subjacente para executar o processo de reconfiguração. Um dos primeiros trabalhos sobre reconfiguração dinâmica foi feito por Kramer e Magee (1985). Eles definiram os requisitos necessários para o suporte à reconfiguração dinâmica. Também identificaram que, tanto a linguagem de programação, quanto a plataforma de execução, devem fornecer suporte e serviços para possibilitar a reconfiguração dinâmica.

O processo tradicional de reconfiguração de um sistema em operação consiste em: parar o sistema e carregar uma nova configuração, a qual foi gerada a partir de uma compilação. A Figura 21a mostra esse método estático, no qual todos os componentes do sistema são configurados ao mesmo tempo, somente após o carregamento o sistema estará pronto para reiniciar.

O processo de configuração dinâmica, ilustrado na Figura 21b, fornece a possibilidade de modificar ou estender o sistema para atender cenários imprevistos, sem a necessidade de compilar todo o sistema. Tais modificações são feitas sem desligar ou parar o sistema.

Inicialmente, o sistema executa a especificação de configuração inicial. Num certo momento, uma alteração dessa configuração é planejada, ela deve ser validada para garantir que é compatível com sistema atual. Além disso, deve ser planejada a execução dessa reconfiguração, a qual pode introduzir novos componentes, modificar os existentes e fornecer uma interconexão diferente entre esses componentes. Nos sistemas flexíveis ou reconfiguráveis, a reconfiguração dinâmica pode alterar estruturas lógicas e físicas do sistema.

Figura 21 – Processo de configuração estática em (a) e dinâmica em (b)



Fonte: Adaptado de Kramer e Magee (1985)

Esta tese propõe um método para execução desse processo de reconfiguração dinâmica. Aqui, pode-se destacar que o método realiza o planejamento de forma sistemática, baseando-se na configuração atual e a nova configuração do sistema.

4.1 ANÁLISE DOS TRABALHOS RELACIONADOS

Existem vários trabalhos que tratam a reconfiguração dinâmica (RD) com a norma IEC 61499, dentre os quais pode-se citar Olsen et al. (2005), Zoitl, Sunder e Terzic (2006), Almeida, Luntz e Tilbury (2007), Guellouz et al. (2016). Em geral, o requisito básico da RD é a manutenção da consistência do SMR. O que significa mantê-lo operando sem erros.

Alguns trabalhos discutem a utilização de um tipo especial de aplicação de reconfiguração dinâmica (ARD) (ALMEIDA; LUNTZ; TILBURY, 2007; SüNDER; VYATKIN; ZOITL, 2013). Essa aplicação é desenvolvida em paralelo ao sistema de manufatura reconfigurável (SMR) que está em funcionamento e é carregada em tempo de de execução. Esse tipo de solução pode aparecer com diferentes nomes na literatura, como por exemplo *Evolution Control Application* (ECA). Dentre as vantagens, a ARD consegue manter o fluxo correto de execução da aplicação, assim como os seus dados. Ela também identifica o momento correto de se aplicar a alteração desejada. Porém, A ARD exige memória extra do dispositivo, pois ela deve ser carregada e executada pelo mesmo (SüNDER; VYATKIN; ZOITL, 2013).

Apesar da norma IEC 61499 ter fornecido os comandos de RD para os blocos de funções, isso não é suficiente para a configurabilidade (ALMEIDA; LUNTZ; TILBURY, 2007; Guellouz et al., 2016). A intervenção manual ainda é necessária para resolver problemas específicos durante a RD (SüNDER; VYATKIN; ZOITL, 2013; STRASSER et al., 2014). Por exemplo, Strasser et al. (2014) apresentam um estudo de caso sobre RD de um sistema de

manufatura reconfigurável (SMR). O estudo consistia em substituir os FBs com o objetivo de melhorar o algoritmo de controle do SMR em tempo de execução. Eles demonstraram que um planejamento cuidadoso é necessário para desenvolver a ARD, o que requer conhecimentos detalhados do SMR. Eles também identificaram que, mesmo para executar uma RD simples, uma ARD relativamente grande é necessária.

A falta de um modelo formal para os elementos da norma impede que a mesma tenha adoção nas práticas da indústria (DUBININ; VYATKIN; SHALYTO, 2016; ATMOJO; SALCIC; WANG, 2018). Várias técnicas de modelagem formal já foram propostas para a IEC 61499, em particular para os blocos de função e o seu ECC (PANG; VYATKIN, 2007). Blech et al. (2016) apresentam uma visão geral de técnicas de modelagem e verificação para a IEC 61499. A Tabela 3 mostra algumas abordagens para a modelagem dos elementos da IEC 61499.

Tabela 3 – Abordagens de Modelagem Formal para a IEC 61499.

Autores	Abordagem	Ferramentas
Vyatkin e Hanisch (2001)	NCES / SNS	VEDA
Wurmus e Wagner (2000)	CNet	-
Frey e Hussain (2006)	Timed Automata	UPPAL
Zhang, Diedrich e Halang (2004)	FSM	-
Dubinin, Vyatkin e Shalyto (2016)	TM	-

Fonte: Adaptado de (FREY; HUSSAIN, 2006)

Um dos primeiros trabalhos em modelagem formal para a IEC 61499 foi proposto por Vyatkin e Hanisch (2001). Eles apresentaram uma abordagem e uma ferramenta de software para incorporar a verificação formal às práticas da engenharia de controle. A ferramenta, chamada de VEDA (*Verification Environment for Distributed Applications*), foi desenvolvida para simulação baseada em modelo e verificação integrados numa interface gráfica. A modelagem dos elementos da norma foi feita através do formalismo NCES (*Net Condition Event Systems*) (RAUSCH; HANISCH, 1995), que é baseado em sistemas Condição/Evento, desenvolvido por Sreenivas e Krogh (1991). Eles mostraram que um modelo em NCES pode ser gerado a partir de um sistema aderente a IEC 61499 e verificado.

Em um artigo posterior, Pang e Vyatkin (2007) demonstraram a modelagem de vários elementos da IEC 61499 utilizando NCES. Essa abordagem modela o FB em termos de 5 máquinas de estados diferentes, as quais lidam com eventos e dados de entrada, ECC e eventos e dados de saída. Porém, segundo os autores, a modelagem completa dos algoritmos internos dos FBs é impossível ou muito complexa. Portanto, eles optaram por modelar somente a saída dos algoritmos.

Sünder, Vyatkin e Zoitl (2013) mostraram a modelagem e verificação por NCES da execução de uma reconfiguração dinâmica num exemplo prático e simples. Nesse artigo, eles utilizaram a ferramenta ViVe (*Visual Verification toolset*). A reconfiguração dinâmica ocorre por meio da execução de uma ECA, a qual é feita sob medida para cada aplicação de controle

que se deseja modificar. Com a ferramenta ViVe, eles mostraram que foi possível verificar por *model-checking* a execução da reconfiguração dinâmica nesse sistema.

Segundo Frey e Hussain (2006), Wurmus e Wagner (2000) apresentaram um formalismo baseado em redes de Petri, chamado CNet. O modelo proposto lida somente com eventos e propõe que os dados sejam integrados utilizando uma variante da rede de Petri chamada *Signal Interpreted Petri Net* (SIPN). Mais tarde, Hagge e Wagner (2005) formularam um modelo da IEC 61499 em termos de CNet (*Component Net*). Segundo Frey e Hussain (2006), a modelagem em CNet não é estritamente correspondente ao ECC e, devido ao encapsulamento de dados em *tokens*, requer elementos adicionais para realizar conexões entre FBs.

Uma modelagem utilizando autômato temporizado é citada por Frey e Hussain (2006). Foi introduzido um conceito modular que atribui para cada FB um conjunto de autômatos temporizados. O modelo abstrai o fluxo de dados da rede de FBs e, consequentemente, abstrai as transições do ECC que envolvem condições de guarda com variáveis, assim como os algoritmos (PANG; VYATKIN, 2008).

Zhang, Diedrich e Halang (2004) propõem a modelagem da IEC 61499 utilizando AFD. Nesse trabalho, os autores propõem modelar os FBs em AFD para executar verificações utilizando ferramentas baseadas em AFD. Porém, o conceito proposto desconsidera a execução de múltiplos algoritmos e o acionamento de ECCs simultaneamente.

Dubinin, Vyatkin e Shalyto (2016) propõem a utilização de *Transition Systems* (TS) para formalizar a semântica dos blocos de função da IEC 61499. Segundo os autores, a proposta é mais conveniente para a verificação formal de sistemas de blocos de funções. Eles desenvolveram um método para codificar um sistema com FBs em modelos para a ferramenta de *model-checking* SMV. Eles pretendem desenvolver ferramentas de software para realizar essa conversão automaticamente.

Guellouz et al. (2016) propuseram uma extensão para a norma IEC 61499 com blocos de função reconfiguráveis (RFBs). Os RFBs são projetados pelo usuário a fim de operar em mais de um modo, os quais são previstos em tempo de projeto. Logo, o RFB possui um ECC para cada possível forma de execução do mesmo. O RFB é composto por um ECC controlador, ECCs escravos, eventos de reconfiguração e dados de reconfiguração, além dos já conhecidos eventos, dados e algoritmos do FB atual. De acordo com o modo de operação selecionado através de um evento de reconfiguração recebido, o ECC controlador seleciona um dos ECCs escravos para operação.

Atmojo et al. (2018) desenvolveram uma linha de montagem flexível capaz de produzir produtos customizados de acordo com a demanda. Eles estenderam a planta didática ENAS para que a mesma pudesse produzir dois tipos de produtos distintos. De acordo com a demanda, a planta passa a produzir um ou outro tipo de produto. A abordagem permite a reconfiguração do

sistema em tempo de execução. Porém, a reconfiguração é prevista no projeto.

Na área de sistemas a eventos discretos (SEDs), Macktoobian e Wonham (2017) desenvolveram um método para a reconfiguração dinâmica de SEDs baseado na TCS. O método permite trocar o modo de operação do sistema em tempo de execução, de forma que a consistência do mesmo seja garantida. Entretanto, os modos de operação são pré-determinados já no projeto do sistema, ou seja, não trata de reconfigurações não previstas em projeto.

Sampath et al. (2008) apresentaram uma estratégia baseada em redes de Petri para a reconfiguração dinâmica de SEDs, a qual considera somente eventos controláveis. Eles também apresentam um estudo de caso, no qual o método proposto controla o gerenciamento de recursos de um hospital dinamicamente. O fato de considerar apenas eventos controláveis torna essa abordagem difícil de ser aplicada em SMR, pois nestes há eventos não controláveis provenientes de sensores por exemplo.

Nahabedian et al. (2018) apresenta uma estratégia de reconfiguração dinâmica para SMR baseada em autômatos, a qual permite alterar a lógica de controle em tempo de execução. A abordagem gera uma série de estados intermediários que conduzem o sistema de uma configuração inicial para uma nova. Os autores apresentam uma formulação precisa do método e um dos casos de estudo é uma célula de produção. Eles levantam questões cujo o método proposto não pôde resolver, como por exemplo: Quando a lógica de controle (LC) pode executar a RD? Quando novas máquinas do sistema, por exemplo, devem ser inciadas? Quando uma máquina pode ser removida? Essas questões foram resolvidas pelo método proposto nesta tese, o qual será apresentado no Capítulo 5.

4.2 CONSIDERAÇÕES DO CAPÍTULO

Esse Capítulo apresentou a definição geral de RD definida por Kramer e Magee (1985). Essa definição ainda é utilizada nos trabalhos recentes na área de RD. Segundo essa definição, para possibilitar a RD o SMR deve fornecer suporte a execução de comandos que possam acessar e modificar a lógica de controle (LC). Faz-se então uma distinção entre reconfiguração tradicional, a qual exige o desligamento do sistema para possibilitar a atualização, e a RD, que é executada em tempo de execução.

Executar uma RD num SMR levanta questões como as apontadas por (NAHABEDIAN et al., 2018). Na literatura consultada, nota-se que ainda existem lacunas, as quais são tratadas nesta tese, que são: Quando a nova LC deve tomar o controle do sistema? Como e quando remover, adicionar, atualizar ou substituir uma máquina ou robô. Quais operações o sistema pode continuar executando durante a RD? Como isolar as regiões críticas da RD? Nota-se ainda, que a RD desta tese trata de modificações não previstas no projeto, diferente das soluções de Atmojo et al. (2018) e (MACKTOOBIAN; WONHAM, 2017) por exemplo.

Em alguns dos trabalhos relacionados a IEC 61499 apresentados, essas questões são

respondidas pelo uso de uma ARD. Essa é projetada para reconfigurar o sistema no momento correto. Todos os passos da RD são previamente definidos e implementados na ARD. Essa abordagem consome um tempo considerável e exige conhecimentos detalhados da aplicação.

Para desenvolver uma ARD usa-se a configuração atual do sistema e a nova, a partir de então os passos da RD são definidos e implementados. O método proposto nesta tese, que é apresentado no Capítulo 5, se diferencia desta abordagem porque define os passos da RD através de um processo sistemático. Além disso, outro objetivo do método proposto é que a solução seja correta por construção, o que significa eliminar o processo de validação apresentado na Figura 21b. Quando tal método é utilizado com a IEC 61499, a ARD pode ser gerada automaticamente, isso é apresentado no Capítulo 6.

Assim, o método de RD proposto nesta tese também trata a principal limitação identificada na literatura consultada, a qual consiste em alterar a parte física do sistema de forma sistemática. A maior parte dos trabalhos de RD não suporta ou não apresenta um estudo de caso para demonstrar reconfiguração da parte física do sistema de forma automática. O Capítulo 6 apresenta um segundo método proposto nesta tese a fim de realizar a automatização do processo de RD.

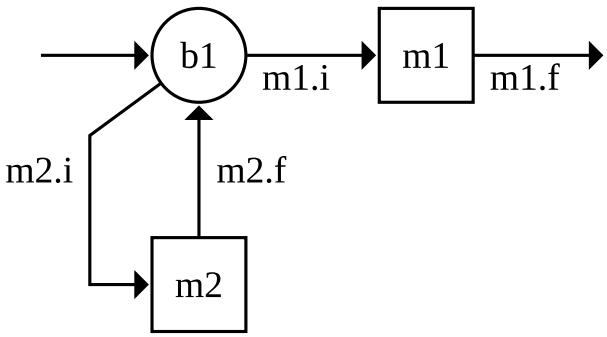
5 RECONFIGURAÇÃO DINÂMICA SEGURA

Este capítulo apresenta o método de reconfiguração dinâmica segura (RDS), que é a principal contribuição desta tese. A RDS permite executar uma RD a fim de modificar um sistema de manufatura reconfigurável (SMR) em relação à sua planta e lógica de controle (LC). Assim, o SMR pode atender a novos requisitos não previstos no seu projeto anterior. A RDS também pode ser utilizada para isolar partes do SMR para fins de manutenção e atualização.

Uma LC é representada por um autômato finito determinístico (AFD), que é composto por estados e transições. O ponto chave da RDS é a premissa de que qualquer LC pode ser substituída por outra no seu estado inicial. A reconfiguração em sistemas de manufatura tradicionais se dá através do desligamento, execução das modificações e reinicio do sistema a partir de seu estado inicial. Como este representa o sistema num estagio no qual não há nenhuma tarefa em execução, então qualquer modificação poderá ser realizada.

Por exemplo, considerando um SMR que produza uma peça com apenas uma máquina $m1$, deseja-se fazer uma alteração para incluir uma nova máquina $m2$ que faça um trabalho nessa peça antes que ela vá para a máquina $m1$. A Figura 22 ilustra a planta com as duas máquinas. Considera-se aqui que a máquina $m1$ não será modificada e nem será preciso mudá-la de local físico para a instalação de $m2$, a qual executa um processo antes de $m1$. A Figura 23a ilustra a LC C_1 atual e a 23b ilustra a nova LC C_2 desejada. O evento controlável $m1.i$ inicia a operação da máquina $m1$, enquanto o evento não controlável $m1.f$ indica que a mesma terminou sua operação. A máquina $m2$ funciona de forma análoga com os eventos $m2.i$ e $m2.f$. Primeiro a máquina $m2$ retira a peça do *buffer* $b1$, faz um processo e a devolve para o mesmo *buffer*. Depois, a máquina $m1$ executa o seu processo. Somente após a operação de $m1$ uma nova peça é colocada como entrada do sistema no *buffer* P.

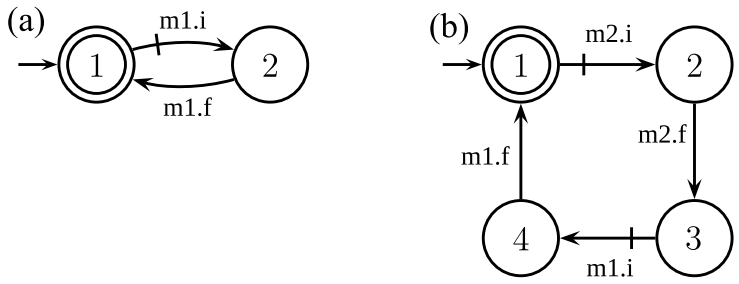
Figura 22 – Planta com as duas máquina $m1$ e $m2$. A máquina $m2$ é inserida no processo por meio de uma RD.



Fonte: Próprio Autor

Na reconfiguração tradicional, desliga-se o sistema, faz-se as alterações e reinicia-o no estado inicial. Uma forma mais dinâmica seria substituir C_1 por C_2 no momento em que C_1

Figura 23 – (a) mostra C_1 com apenas a máquina $m1$ e (b) mostra C_2 com a nova máquina $m2$ operando antes de $m1$.



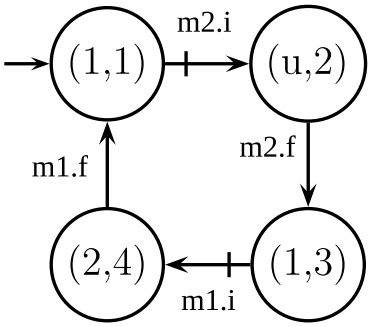
Fonte: Próprio Autor

atingisse o estado 1 e impedir que C_2 dispare $m2.i$ até que a máquina $m2$ esteja instalada. Neste exemplo, apenas por observação conclui-se que o estado 1 é um estado seguro para executar essa RD. Mesmo para sistemas mais complexos, a RD pode ser executada desta forma. Entretanto, a passagem pelo estado 1 é equivalente a reiniciar o SMR. Em exemplos mais complexos, que apresentam atividades em paralelo, raramente o sistema retorna ao estado inicial. O SMR poderia de certa forma ser forçado a passar pelo estado inicial, mas como mencionado, é o mesmo que reiniciar o sistema. Pode-se concluir que o estado inicial é um estado equivalente entre duas LCs quaisquer, pois representa o sistema num estágio completamente parado, no qual os elementos do mesmo, como máquinas, ainda começarão a atuar.

O conceito de "estados equivalentes" é igualmente importante para a RDS. Eles representam as sequências de operações que o sistema pode continuar executando durante o processo de RD. Por isso, é importante que o método da RDS foque em localizar o maior número possível de estados equivalentes entre as duas LCs, além do estado inicial. Isto é uma das funções do algoritmo $Rutt$, que foi desenvolvido nesta tese. O SMR permanecerá executando nos estados equivalentes enquanto, por exemplo, uma nova máquina é instalada, atualizada ou removida. Pode-se dizer ainda, que os estados equivalentes representam as características de controle que permanecerão as mesmas antes e depois da RD, que são as partes da LC que controlam máquinas ou obedecem especificações de controle que não serão modificadas.

O algoritmo $Rutt$ recebe como entrada as LCs C_1 e C_2 e gera a LC $C_{1\to2}$. A Figura 24 é a solução do algoritmo $Rutt$ da RDS para este problema de RD. Nota-se que os rótulos de cada estado são um par, no qual o primeiro elemento representa um estado de C_1 e o segundo um estado de C_2. A marcação u no estado $(u, 2)$ indica que não há estado equivalente a este em C_1. Todos os estados, cujos rótulos não apresentam a marcação u, são estados equivalentes. Observa-se que, transitando apenas pelos estados equivalentes, a única ação possível é o acionamento da máquina $m1$. Em resumo, o algoritmo $Rutt$ gera uma LC que é uma combinação das duas LCs de entrada. Serão estados equivalentes aqueles que existem em ambas as LCs.

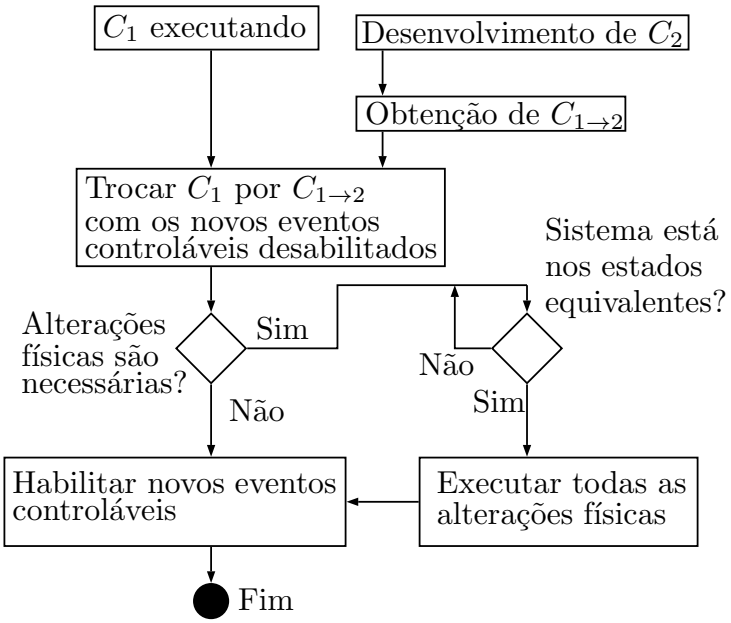
Figura 24 – A LC $C_{1\to2}$, que é o retorno do Algoritmo Rutt ao receber C_1 e C_2 como entrada.



Fonte: Próprio Autor

O método da RDS é ilustrado na Figura 25. Tendo o SMR sob o controle da LC C_1. E ainda, tendo a LC C_2. O próximo passo é a obtenção da LC $C_{1\to2}$, a qual é realizada com o algoritmo $Rutt$. Serão utilizadas como exemplos as LCs C_1, C_2 e $C_{1\to2}$ já apresentadas.

Figura 25 – Fluxo de execução da RDS.



Fonte: Próprio Autor

Pode-se, então, realizar a troca de C_1 por $C_{1\to2}$. Uma vantagem da RDS é que tal substituição pode ser realizada a qualquer momento, sem precisar aguardar alguma situação adequada. Isso é possível porque para cada estado de C_1, existe um estado equivalente em $C_{1\to2}$. Portanto, faz-se a troca e copia-se o estado. Por exemplo, se C_1 estava no estado 2 durante a troca, então $C_{1\to2}$ deve partir do estado $(2,4)$. Se o estado de C_1 era 1, então $C_{1\to2}$ pode partir tanto do estado $(1,1)$ quanto $(1,3)$, pois ambos representam a máquina a parada. A troca é realizada mantendo os novos eventos controláveis desabilitados. Neste caso, o evento $m2.i$ não poderá ser executado.

Se não há alterações físicas, os eventos anteriormente desabilitados são habilitados e o sistema estará operando com a nova LC. Quando há alterações físicas, o usuário deve aguardar que o SMR entre nos estados equivalentes, dos quais não poderá mais sair até o final dessas. Uma vez que o SMR está nos estados equivalentes, as alterações físicas são executadas. Ao término destas, os novos eventos controláveis são habilitados.

Este capítulo está organizado da seguinte forma: a Seção 5.1 apresenta o algoritmo $Rutt$, uma definição mais formal para os estados equivalentes e alguns exemplos de processamento pelo mesmo. A Seção 5.2 faz uma comparação com um trabalho similar de RD em SEDs, apresentando as principais diferenças. A Seção 5.3 apresenta um estudo de caso de uma célula de produção com máquinas em paralelo, com o objetivo de evidenciar que o SMR pode continuar operando durante uma RD com o uso da RDS. Este estudo demonstra a importância dos estados não acessíveis no contexto da RDS. A Seção 5.4 apresenta um estudo de caso de uma célula de produção sequencial, na qual altera-se uma máquina no centro da linha. Por fim, a Seção 5.5 apresenta as considerações deste capítulo.

5.1 GERANDO O MODELO $C_{1\to2}$

O principal passo da RDS é a obtenção de $C_{1\to2}$, a qual é feita através do algoritmo *Rutt* (*Runtime Transformation of Transition System*). Assim $C_{1\to2} = Rutt(C_1, C_2)$, onde C_1 é a LC inicial e C_2 é a nova LC. Antes de definir o algoritmo *Rutt*, é necessária a definição do algoritmo *ISOTRIM*, que é uma dependência do algoritmo *Rutt*.

5.1.1 Algoritmo ISOTRIM

O algoritmo *ISOTRIM*, mostrado no Algoritmo 1, percorre um autômato qualquer dado como entrada a partir do estado inicial ignorando a direção das transições. Todos os estados e transições encontrados são então adicionados ao autômato de saída. A Figura 26 mostra um exemplo. O algoritmo *ISOTRIM* remove apenas os estados 4 e 5 e o evento d, pois estão desconectados do estado inicial. As partes do algoritmo *ISOTRIM* são descritas como segue:

A parte 1 do algoritmo *ISOTRIM* percorre o autômato de entrada inserindo em Q' os estados encontrados. Começando com $Q' = \{1\}$, a primeira iteração do comando *enquanto* encontra os estados 6 e 2, então $Q' = \{1, 2, 6\}$. Na próxima iteração obtém-se $Q' = \{1, 2, 3, 6\}$. Finalmente, o comando *enquanto* termina devido a $Q_u = \emptyset$.

A parte 2 do algoritmo *ISOTRIM* gera a nova função de transição (δ'), a qual contém somente transições entre estados de Q'. A parte 3 gera o conjunto de eventos (Σ'), que inclui somente os eventos encontrados na função de transição gerada no passo anterior. A parte 4 define como estados finais em G' aqueles estados que também são finais em G.

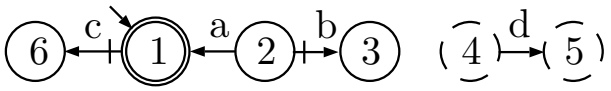
Na teoria dos autômatos, estados que não podem ser acessados a partir do estado inicial são chamados de estados não acessíveis. Eles são considerados inúteis, pois é impossível

Algoritmo 1: $ISOTRIM$ (*Isolated trim*).

Dados: G: Um autômato $(Q, \Sigma, \delta, q_0, M)$
Resultado: G': Um autômato $(Q', \Sigma', \delta', q'_0, M')$

1 *PARTE 1: Somente estados que se conectam com o estado inicial, independentemente da direção da transição*;
2 $Q_u \leftarrow \{q_0\}$;
3 $Q' \leftarrow \{q_0\}$;
4 **enquanto** $Q_u \neq \emptyset$ **faça**
5 $\quad Q_u \leftarrow \{q_x | q_x \in Q \text{ e } q_x \notin Q' \text{ e } \exists q_n.(q_n \in Q') \text{ e } \delta(q_n, \sigma) = q_x \vee \delta(q_x, \sigma) = q_n\}$;
6 $\quad Q' \leftarrow Q' \cup Q_u$;
7 *PARTE 2: Define a função de transição*;
8 $\delta'(q_a, \sigma) = \begin{cases} \delta(q_a, \sigma) & \text{se } q_a \in Q' \text{ e } \delta(q_a, \sigma) \in Q' \\ \text{não definida} \end{cases}$
9 *PARTE 3: Define o alfabeto*;
10 $\Sigma' \leftarrow \{\sigma | \delta'(q_a, \sigma) \text{ é definida e } q_a \in Q'\}$
11 *PARTE 4: Define os estados finais*;
12 $M' \leftarrow \{q | q \in (M \cap Q')\}$

Figura 26 – Resultado do algoritmo $ISOTRIM$. Somente os estados 4 e 5 e o evento d são removidos do autômato original.



Fonte: Próprio Autor

alcançá-los. Adiante nesse capítulo, será demonstrada a importância desses estados no contexto da RD.

5.1.2 O Algoritmo $Rutt$

O modelo $C_{1\to2}$ contém todos os estados de C_1 e C_2. Os estados equivalentes (ou estados seguros para reconfiguração dinâmica) são encontrados pelo algoritmo *Rutt* e mesclados. O algoritmo *Rutt*, mostrado no Algoritmo 2, funciona como segue:

A parte 1 do algoritmo encontra os estados equivalentes. O símbolo u (na linha 2) é utilizado para identificar estado não definido posteriormente. O comando na linha 3 constrói um conjunto de pares com o produto cartesiano de C_1 e C_2. Os conjuntos de eventos são unidos. O estado inicial de $C_{1\to2}$ é o par formado pelo estado inicial de C_1 e C_2. Uma transição com evento σ a partir do estado (q^{C_1}, q^{C_2}) somente existirá se as transições $\delta_1(q^{C_1}, \sigma)$ e $\delta_2(q^{C_2}, \sigma)$ são definidas. Então, *ISOTRIM* é aplicado para remover os estados isolados.

Assim, $C_{1\to2}$ é formado somente por estados equivalentes, o que significa que qualquer sequência de eventos permitida em $C_{1\to2}$ é também permitida tanto em C_1 quanto em C_2. Por

Algoritmo 2: $Rutt$ (*Runtime transformation of transition system*).

Dados: $C_1 = (Q^{C_1}, \Sigma^{C_1}, \delta_1, q_0^{C_1}, M^{C_1})$
Dados: $C_2 = (Q^{C_2}, \Sigma^{C_2}, \delta_2, q_0^{C_2}, M^{C_2})$
Resultado: $C_{1\to 2} = (Q, \Sigma, \delta, q_0, M)$

PARTE 1: Obter os estados equivalentes;
u : representa nulo;
$Q \leftarrow (Q^{C_1} \times Q^{C_2})$;
$\Sigma \leftarrow \Sigma^{C1} \cup \Sigma^{C2}$;
$q_0 \leftarrow (q_0^{C_1}, q_0^{C_2})$;
para cada $(q_a, q_b) \in Q$ **faça**
    $\delta((q_a, q_b), \sigma) = \begin{cases} (\delta_1(q_a, \sigma), \delta_2(q_b, \sigma)) & \text{se ambos são definidos} \\ \text{não definido} & \text{caso contrário} \end{cases}$;
$C_{1\to 2} \leftarrow$ ISOTRIM$(C_{1\to 2})$;
PARTE 2: Desvio para o estado inicial se o estado é da forma $(q^{C_1}, q_0^{C_2})$;
para cada $(q^{C_1}, q^{C_2}) \in Q$ **faça**
    $\delta'((q^{C_1}, q^{C_2}), \sigma) = \begin{cases} (q_0^{C_1}, q_0^{C_2}) & \text{se } \delta_2(q^{C_2}, \sigma) = q_0^{C_2} \\ \delta((q^{C_1}, q^{C_2}), \sigma) & \text{se } \delta_2(q^{C_2}, \sigma) \neq q_0^{C_2} \end{cases}$;
$\delta \leftarrow \delta'$;
$C_{1\to 2} \leftarrow$ ISOTRIM$(C_{1\to 2})$;
PARTE 3: Adiciona os estados não equivalentes de C_1 *e todas as suas transições de saída*;
$Q \leftarrow Q \cup \{(q, u) | q \in Q^{C_1}$ tal que $\nexists q_x.(q, q_x) \in Q$ e $q_x \in Q^{C_2}\}$;
para cada $(q, u) \in Q$ **faça**
    Adicione a seguinte transição em δ:
    $\delta((q, u), \sigma) =$
    $\begin{cases} (\delta_1(q, \sigma), a) & \text{se } \delta_1(q, \sigma) \text{ é definido e existe o estado } (\delta_1(q, \sigma), a) \in Q \\ & \text{e } a \in (Q^{C_2} \cup u) \\ \text{não definido} & \text{caso contrário} \end{cases}$;
PARTE 4: Adiciona estados não equivalentes de C_2 *e todas as suas transições de entrada e saída*;
$Q \leftarrow Q \cup \{(u, q) | q \in Q^{C_2}$ tal que $\nexists q_x.(q_x, q) \in Q$ onde $q_x \in Q^{C_1}\}$;
para cada $(u, q) \in Q$ **faça**
    Adicione a seguinte transição em δ:
    $\delta((u, q), \sigma) =$
    $\begin{cases} (a, \delta_2(q, \sigma)) & \text{se } \delta_2(q, \sigma) \text{ é definido e existe o estado } (a, \delta_2(q, \sigma)) \in Q \\ & \text{e } a \in (Q^{C_1} \cup u) \\ \text{não definido} & \text{caso contrário} \end{cases}$;
    para cada $(q_x, q_y) \in Q$ **faça**
        se $\delta_2(q_y, \sigma) = q$ **então**
            Adicione a seguinte transição em δ:
            $\delta((q_x, q_y), \sigma) = (u, q)$;

exemplo, se em C_1 existe uma transição $(q_0^{C_1}, \sigma, q_1^{C_1})$ e em C_2 existe uma transição $(q_0^{C_2}, \sigma, q_3^{C_2})$, então existirá em $C_{1\to2}$ a transição $((q_0^{C_1}, q_0^{C_2}), \sigma, (q_1^{C_1}, q_3^{C_2}))$.

A parte 2 do algoritmo *Rutt* altera a função de transição para desviar para o estado inicial de $C_{1\to2}$ as transições cujo destino é da forma $(q^{C_1}, q_s^{C_2})$, onde $q_s^{C_2}$ é o estado inicial de C_2. Como um estado isolado pode surgir, *ISOTRIM* é aplicado novamente.

A parte 3 do algoritmo inclui todos os estados de C_1 que ainda não estão em nenhum dos pares de Q. Então, esses estados são adicionados na forma (q^{C_1}, u), o que significa que não há estado equivalente para q^{C_1} em Q^{C_2}. As transições de saída desses estados também são adicionadas. Essas transições podem alcançar estados das formas (q^{C_1}, u) ou (q^{C_1}, q^{C_2}).

A parte 4 do algoritmo *Rutt* é similar à parte 3. Entretanto, os estados de C_2 são adicionados na forma (u, q^{C_2}). Adicionalmente, são incluídas as transições de entrada dos estados, além das de saída.

O modelo C_1 pode ser substituído a qualquer momento por $C_{1\to2}$, pois este tem todos os estados do anterior. O processo de RD deve apenas manter o mesmo estado. Por exemplo, se C_1 estava no estado q_n, após a RD, então $C_{1\to2}$ deve partir do estado (q_n, x), onde $x \in Q^{C_2} \cup \{u\}$.

Apesar do Algoritmo $Rutt$ não ter uma prova formal, pode-se argumentar em favor da sua corretude. Cada passo do algoritmo foi elaborado de forma que o autômato resultante $C_{1\to2}$ reconheça somente sequências de eventos especificadas por C_1 ou C_2. A PARTE 1 encontra sequências de eventos válidas tanto em C_1 quanto em C_2. A PARTE 2 apenas desvia para o estado inicial transições cujo destino em C_2 é o estado inicial e em C_1 é outro estado qualquer. A PARTE 3 adiciona as sequências de eventos válidas e existentes somente em C_1. Enquanto a PARTE 4 adiciona as sequências de eventos válidas e existentes somente em C_2. Portanto, o autômato $C_{1\to2}$ comporta todas as possíveis sequências de eventos de C_1 e C_2. Na execução, as sequências pertencentes somente a C_1 serão eliminadas em algum momento, pois é um dos objetivos da nova LC C_2.

5.1.3 Definição de Estado Equivalente

Pode-se agora definir os estados equivalentes como sendo os estados de $C_{1\to2}$ da forma (q_n, q_m), onde $q_n \in Q^{C_1}$ e $q_m \in Q^{C_2}$. Ou seja, o rótulo de um estado equivalente não contém u.

Os estados equivalentes permitem que o sistema se mantenha em execução durante o processo de RD, principalmente quando há alterações físicas. Neste caso, o sistema se mantém nesses estados até que as alterações físicas sejam completadas.

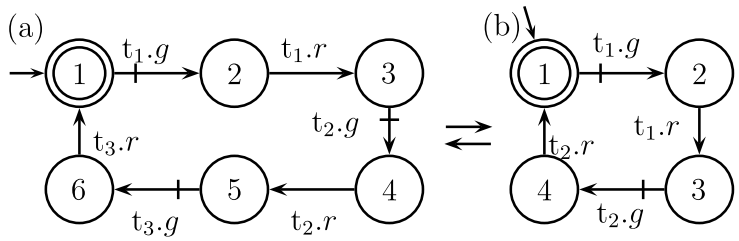
5.1.4 Exemplo de Execução do Algoritmo Rutt

Para exemplificar o funcionamento do algoritmo $Rutt$, será utilizado um estudo de caso similar ao trabalho de Pinto et al. (2016), o qual trata de um sistema de semáforos. O

problema de RD relatado por Pinto et al. (2016) refere-se ao fato de que, ao remover a parte do software que controlava um dos semáforos, este poderia permanecer travado com a luz verde acesa, possibilitando que mais de um semáforo indicasse passagem liberada ao mesmo tempo. O método da RDS resolve este problema gerando uma LC que fecha o semáforo antes dele ser removido. O estudo de caso de Pinto et al. (2016) foi simplificado para facilitar o entendimento do algoritmo $Rutt$, mas isso não diminui a complexidade do problema de RD apontado.

O estudo de caso desta seção consiste num sistema de semáforos (SF) com três semáforos t_1, t_2 e t_3. Cada um com um conjunto de duas lâmpadas de cores verde (g) e vermelho (r). A cor verde só pode ser apresentada por um semáforo de cada vez. Ao ligar, cada semáforo estará sinalizando vermelho. Assim, a LC C_1 para este SF é apresentada na Figura 27a. O evento $t_1.g$ coloca o primeiro semáforo em verde. Após um período, este dispara o evento $t_1.r$, indicando ao controlador que retornou ao estado em que sua lâmpada vermelha está acionada. Os demais semáforos funcionam de forma análoga.

Figura 27 – Em (a) a LC inicial do sistema. Em (b) a nova LC.



Fonte: Próprio Autor

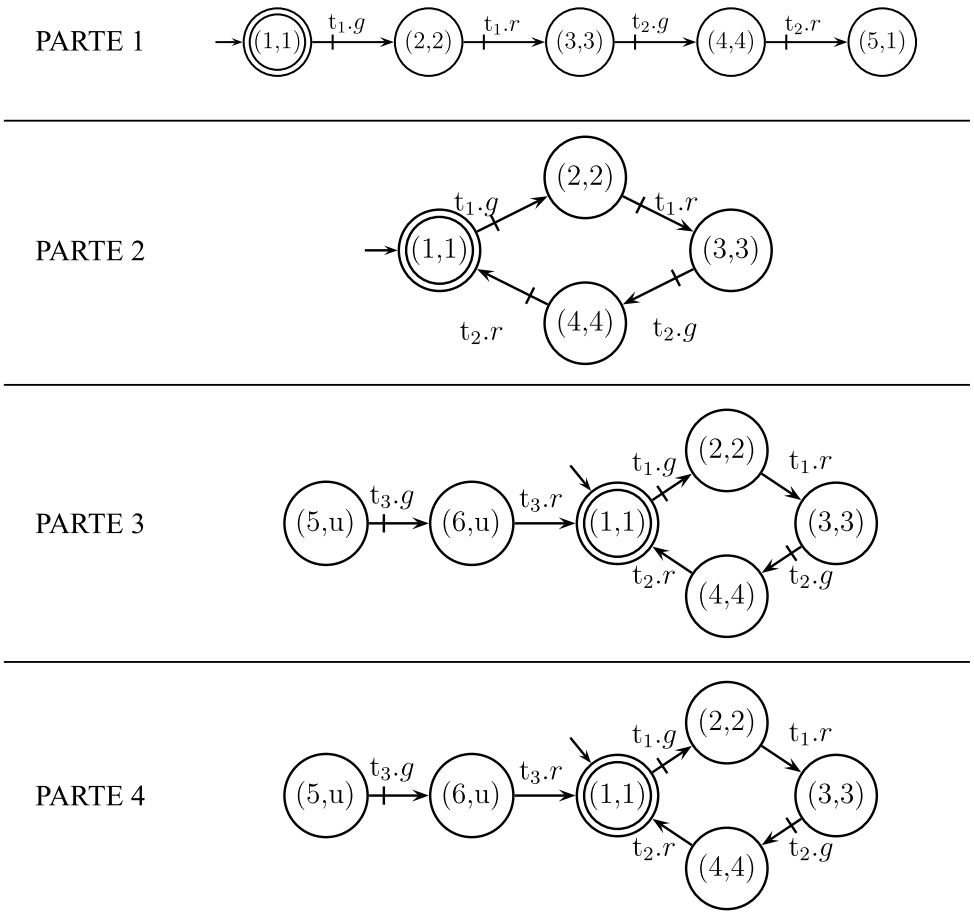
Supõe-se então que uma nova configuração é desejada para o SF, cuja LC C_2 é mostrada na Figura 27b. Esta consiste em remover o semáforo t_3. Faz-se então $C_{1\to2} = Rutt(C_1, C_2)$ para obter a LC $C_{1\to2}$.

A Figura 28 apresenta o resultado do processamento de cada parte do algoritmo $Rutt$ para essas LCs. A parte 1 do algoritmo obtém os estados equivalentes e as transições entre eles. Nota-se que as transições obtidas existem tanto em C_1 quanto em C_2. No estado 4, ambas as LCs possuem uma transição com o evento $t_2.r$, mas para estados diferentes. Denota-se δ_1, δ_2 e $\delta_{1\to2}$ as funções de transição de C_1, C_2 e $C_{1\to2}$ respectivamente. Como $\delta_1(4, t_2.r) = 5$ e $\delta_2(4, t_2.r) = 1$, tem-se $\delta_{1\to2}((4,4), t_2.r) = (5,1)$.

A parte 2 tem como função eliminar estados que são equivalentes ao estado inicial de C_2. Neste caso, o estado $(5,1)$ é um exemplo. Este estado é eliminado e as transições de entrada são desviadas para o estado inicial de $C_{1\to2}$.

A parte 3 inclui os estados de C_1 que não foram considerados equivalentes na parte 1. Além disso, são incluídas todas as suas transições de saída. A parte 4 faz o mesmo procedimento com os estados de C_2, mas todas as transições são incluídas. Neste caso, como C_2 não

Figura 28 – Processamento de cada parte do algoritmo sobre as LCs do sistema de semáforos.



Fonte: Próprio Autor

acrescenta novas especificações, apenas remove, nada foi acrescentado. A PARTE 4 da Figura 28 representa a LC $C_{1\to2}$.

No estudo apresentado por Pinto et al. (2016), o problema ocorre quando a RD é executada no momento em que o semáforo está sinalizando verde. Logo, o método de RDS será simulado na mesma condição, o que significa que o SF está no estado 6 de C_1 (Figura 27a).

No processo da RDS, faz-se a substituição de C_1 por $C_{1\to2}$ mantendo-se o estado. Assim, $C_{1\to2}$ assume o controle do SF no estado $(6, u)$. Para de fato remover o semáforo fisicamente, deve-se esperar o $C_{1\to2}$ alcançar os estados equivalentes, que neste caso são $(1, 1)$, $(2, 2)$, $(3, 3)$, $(4, 4)$. Uma vez nestes estados, o semáforo t_3 estará sinalizando vermelho.

Um segundo estudo de caso pode ser enunciado da seguinte forma: Tendo o SF com 3 semáforos, deseja-se que t_3 permaneça fechado (sinalizando vermelho). A mesma solução pode ser aplicada a este problema, pois ela garante que t_3 será fechado.

Outra analogia possível é: Sendo um SMR formado por 3 máquinas, uma de corte t_1, uma de polimento t_2 e uma de pintura t_3, deseja-se remover a máquina de pintura. Novamente a

mesma solução poderá ser utilizada. Interpreta-se que os estados $(5, u)$ e $(6, u)$ garantem que a máquina de pintura estará vazia e não será mais acionada quando o sistema estiver nos estados equivalentes.

5.2 EXEMPLO DE UMA CÉLULA DE PRODUÇÃO

O estudo de caso apresentado nesta seção foi proposto no trabalho de Nahabedian et al. (2018), cujo o tema também é RD. O estudo trata de uma célula de produção sequencial com duas máquinas, sendo uma delas de polimento inicialmente. A RD desejada é a substituição da máquina de polimento por uma de pintura.

A LC inicial da célula é mostrada por C_G na Figura 29. As peças entram na linha de produção através de um comando in. O controlador dispara $drill$ para iniciar o processo de furação. Os eventos $drillOk$ e $drillNOk$ indicam o término de operação com sucesso e falha respectivamente. Na sequência, dispara-se $polish$ para iniciar a máquina de polimento. Novamente, os eventos $polishOk$ e $polishNOk$ indicam o término de operação com sucesso e falha respectivamente. O evento out indica que a peça saiu da linha de produção. Ainda na Figura 29, $C_{G'}$ é a nova lógica de controle desejada.

O método de Nahabedian et al. (2018) utiliza como entrada a LC atual e suas especificações, requisitos de transição, as novas especificações e um mapeamento entre os estados da atual e a nova LC. No trabalho de Nahabedian et al. (2018), assim como a RDS desta tese, gera-se uma LC que pode substituir a atual em qualquer ponto de execução. A diferença está na forma de condução de uma configuração para outra. Isso é mostrado nos estados intermediários da Figura 29. Nahabedian et al. (2018) utilizam um evento $hotSwap$ para iniciar a condução. O evento $stopOldSpec$ é utilizado para indicar que as antigas especificações deixaram de ser tratadas. O evento $startNewSpec$ é utilizado para indicar que as novas especificações estão em execução. E o evento $reconfigure$, segundo os autores, garante a substituição segura da máquina de polimento pela de pintura.

Para o algoritmo $Rutt$, a entrada é somente a LC atual e a nova que, neste exemplo, são mostradas na Figura 30. A LC gerada pelo algoritmo $Rutt$ é mostrada na Figura 31. A principal diferença é que não há estados extras para a condução de uma configuração para outra. Ao invés disso, o rótulo de cada estado identifica se ele existe na configuração anterior e/ou na nova. E os estados na forma (n, u) (com u na segunda posição do par) indicam antigas especificações.

Um ponto interessante de comparação é executar a troca das LCs no instante logo após o disparo de $polish$. Na Figura 29, essa condição é no estado 22. Nota-se que, quando disparado $hotSwap$, todos os possíveis caminhos levam ao estado 8. Nesse estado, pode-se executar o evento $paint$. Entretanto, um dos caminhos ($hotSwap \rightarrow stopOldSpec \rightarrow startNewSpec \rightarrow reconfigure$) não trata os eventos de resposta da máquina de polimento ($polishOk$ e $polishNOk$). Nota-se que, independente do resultado da máquina de polimento,

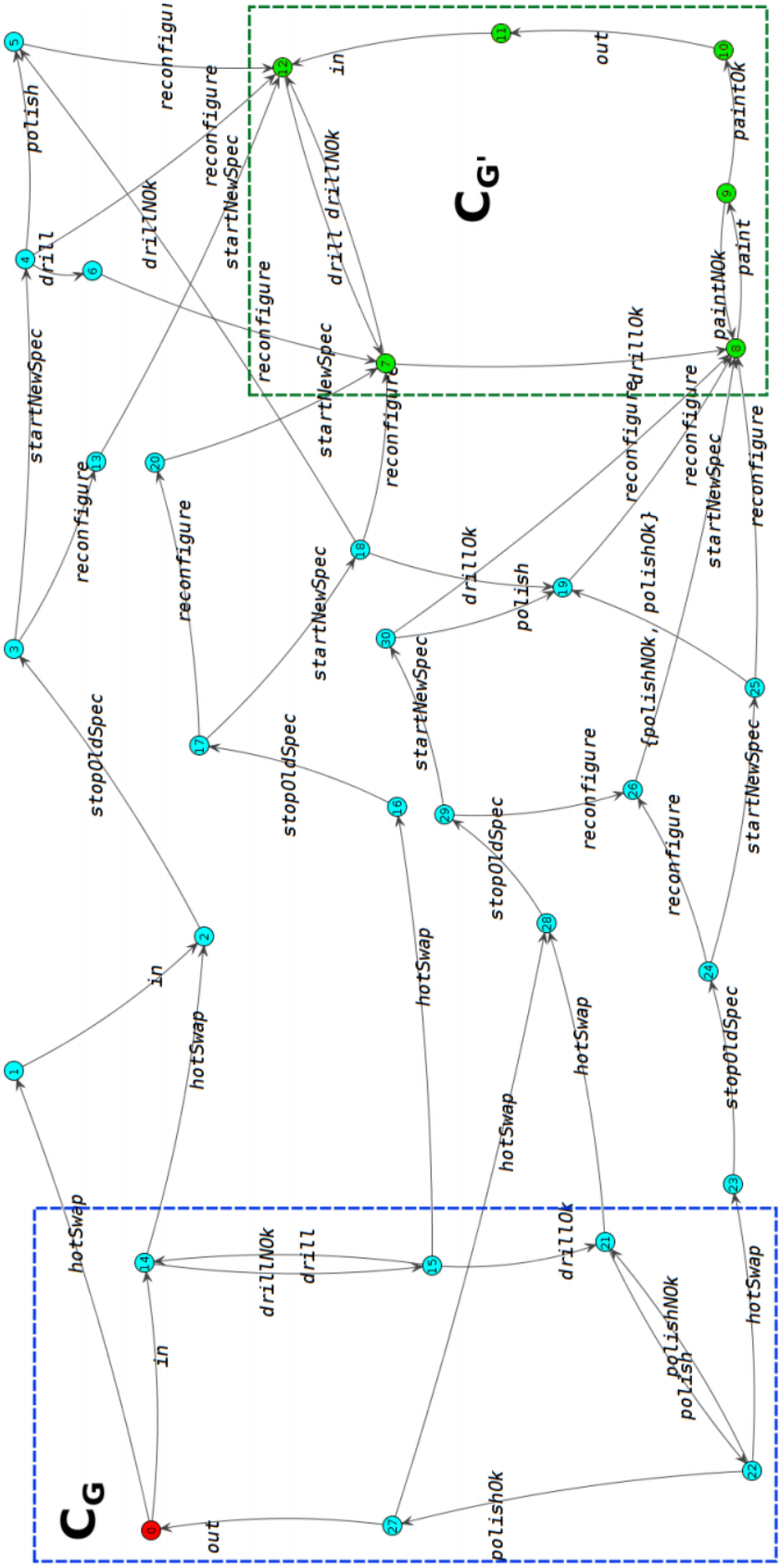
o evento $paint$ será disparado. Segundo os autores, as peças cujo polimento falhou são descartadas, mas não foi apresentado como isso é realizado na prática e porquê o evento $paint$ é disparado sem um novo evento $drill$.

Na LC gerada pelo algoritmo $Rutt$, a condição levantada é no estado $(5, u)$ (Figura 31). O resultado da máquina de polimento será sempre tratado. Se for $polishOk$, a peça será enviada para a saída da linha. Se for $polishNOk$, a peça retorna e o evento $paint$ será disparado.

Isso evidencia uma limitação quando há troca de uma máquina, que possui função de retrabalho, com o objetivo de modificar o produto a ser produzido. Em caso de falha, como $polishNOk$, a nova LC irá retrabalhar a peça executando $paint$. Isso pode ou não ser aceitável dependendo do SMR. Pode uma peça com falha de polimento ser pintada, sendo que a nova configuração não exige polimento? A RDS apresenta a vantagem de não executar $paint$ no caso de sucesso no polimento.

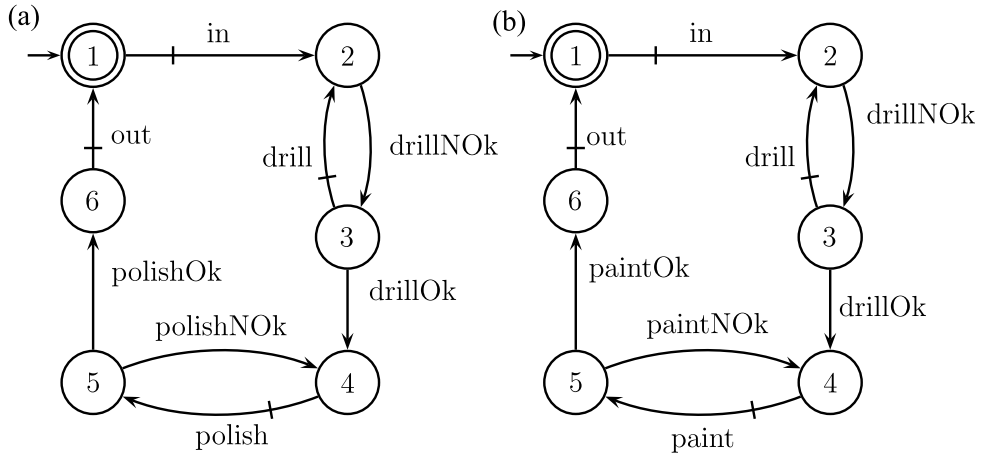
O trabalho de Nahabedian et al. (2018) não apresenta como a substituição das máquina ocorre na prática, logo, não é possível uma comparação nessa questão. Pelo método da RDS, a sequência de operações seria: substitui-se a LC atual pela gerada, aguarda a entrada nos estados equivalentes, faz-se a substituição das máquinas e habilitam-se os novos eventos controláveis ($polish$).

Figura 29 – Controlador de atualização para a célula de produção com três fases. C_G e $C_{G'}$ garantem as especificações antigas e novas respectivamente. Os estados intermediários garantem o progresso da atualização.



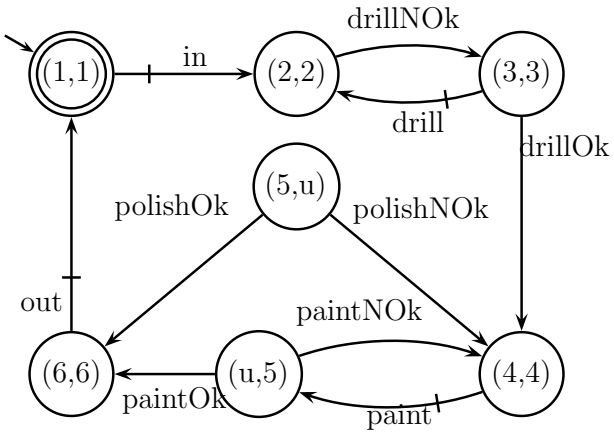
Fonte: (NAHABEDIAN et al., 2018)

Figura 30 – LC anterior e nova LC para a célula de produção de Nahabedian et al. (2018).



Fonte: Próprio Autor

Figura 31 – LC gerada pelo algoritmo *Rutt* para a célula de produção de Nahabedian et al. (2018).



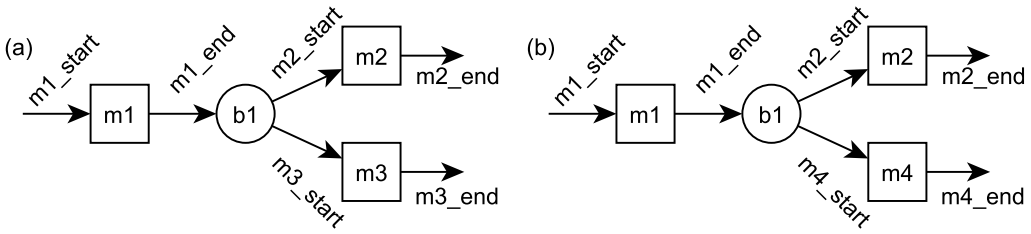
Fonte: Próprio Autor

5.3 EXEMPLO DE UMA CÉLULA DE PRODUÇÃO COM CAMINHO PARALELO

O estudo de caso desta seção tem como objetivo demonstrar que a LC, gerada pelo algoritmo $Rutt$, mantém o SMR operando durante a RD e, se houver uma alternativa, o SMR continuará produzindo. Os outros estudos de caso apresentados são SMR com operação sequencial, logo, qualquer modificação causa um estado no qual não há alternativas de produção, a não ser aguardar que uma máquina, por exemplo, seja instalada.

O SMR deste estudo de caso consiste em 3 máquinas $m1$, $m2$ e $m3$. Cada máquina opera através do disparo do evento mx_start, onde mx é uma das máquinas, e dispara mx_end ao término de sua operação. A Figura 32 mostra esse SMR. A máquina $m1$ deposita peças no *buffer* $b1$, no qual cabe somente uma peça. As máquinas $m2$ e $m3$ pegam as peças de $b1$, fazem as suas respectivas operações e as depositam a frente na célula de produção.

Figura 32 – Planta do SMR com caminho paralelo.



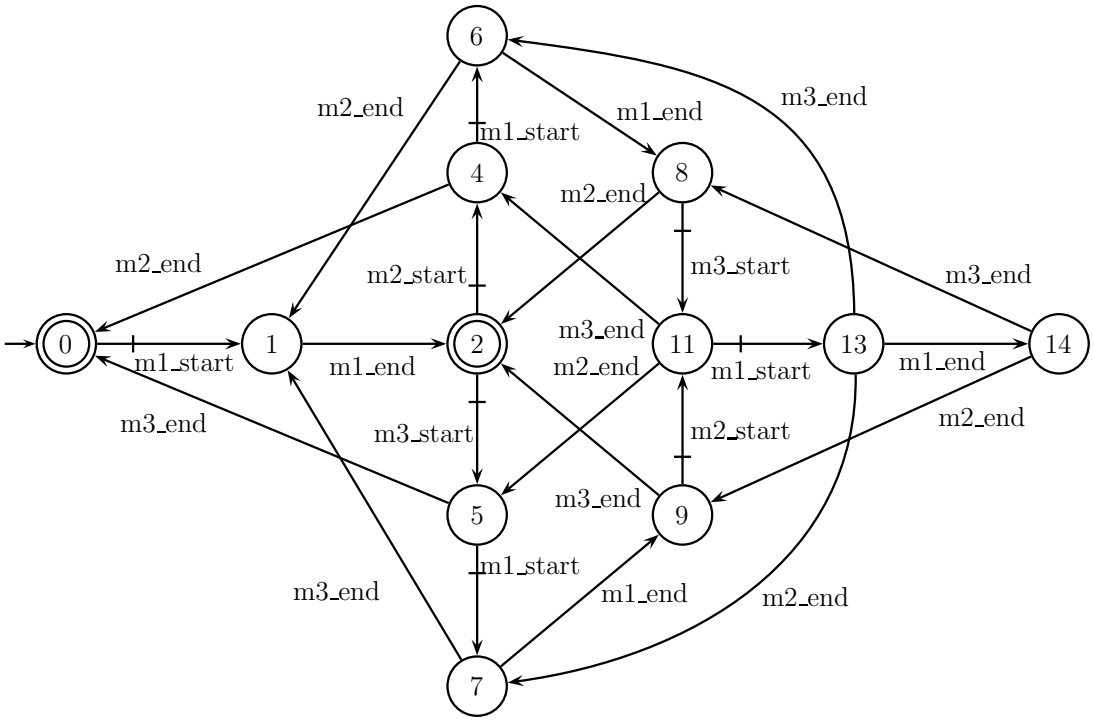
Fonte: Próprio Autor

As especificações desse SMR são feitas em torno do *buffer* $b1$, que são: (a) $m2$ e $m3$ não podem operar antes do término de operação de $m1$; (b) Para cada término de operação de $m1$, somente uma das máquinas $m2$ ou $m3$ podem iniciar operação; (c) $m1$ não pode iniciar uma nova operação enquanto $m2$ ou $m3$ não iniciarem operação (que implica em retirar a peça de $b1$). Essa LC C_1 é mostrada na Figura 33.

A RD desejada para esse SMR é substituir a máquina $m3$ por uma máquina $m4$, a qual possui os mesmos eventos $m4_start$ e $m4_end$. Na prática, isso pode ser feito para colocar no sistema uma máquina mais atualizada, por exemplo. Para isso, a RD deve fornecer uma forma segura de remover a máquina $m3$, nesse caso considera-se aceitável que a máquina deve estar vazia, ou seja, não remover junto uma peça da linha de produção, e não deve mais ser acionada pela LC. Logo, a nova LC C_2 é idêntica a atual, mas troca-se $m3_start$ e $m3_end$ por $m4_start$ e $m4_end$ respectivamente.

O resultado do algoritmo $Rutt$ é ilustrado na Figura 34, a qual mostra a LC $C_{1\to2}$. Nota-se, em destaque tracejado, que o evento $m3_end$ causa a entrada nos estados equivalentes. Dentre esses estados, existem ciclos que mantêm em operação as máquina $m1$ e $m2$, como por exemplo $(2,2) \to (4,4) \to (6,6) \to (8,8) \to (2,2)$. Dessa forma, a RDS mantém o sistema

Figura 33 – LC C_1 para a planta com caminho paralelo.



Fonte: Próprio Autor

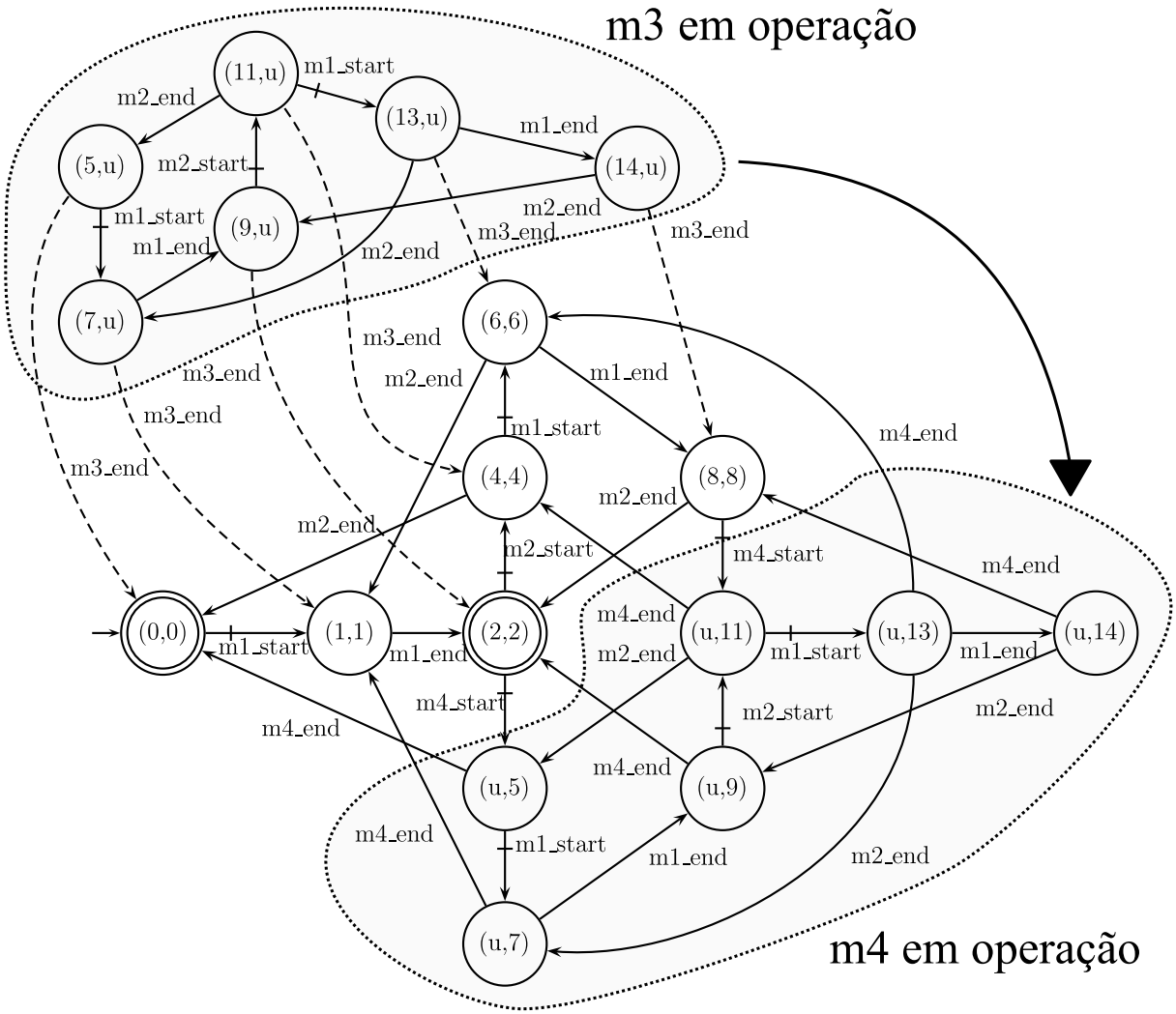
em operação contínua, já que existe mais de um caminho além daqueles que acionam a máquina $m3$.

Quando $C_{1\to2}$ assume o controle do SMR, o evento $m4_start$ está desabilitado. Observa-se, que isso é suficiente para impedir que a LC saia dos estados equivalentes. Nota-se ainda, que a seção na região que representa $m3$ em operação é similar a que representa $m4$ em operação. O algoritmo $Rutt$ isolou essas seções e tornou-as inseguras para a RD, além de fazer a correta remoção da região antiga e inclusão da nova.

Esse estudo de caso mostra uma limitação do processo de RDS evidente na região em que $m3$ está em operação. A existência de ciclos nessa área pode fazer com que a LC nunca alcance os estados equivalentes. Uma possível solução é remover todos as transições com eventos controláveis dessa região, entretanto, isso limitaria a quantidade de operações durante a RD. Por exemplo, no estado $(11, u)$ a única saída seria aguardar o evento $m2_end$. Mas executar $m1_start$ não seria um problema para essa RD, nesse caso estariam $m2$ e $m1$ operando em paralelo enquanto o evento $m3_end$ é aguardado. Outra solução possível é remover apenas o suficiente, das transições com eventos controláveis, para quebrar os ciclos. O problema seria escolher quais das transições remover.

Deve-se observar que a região na qual $m3$ está em operação (Figura 34), compreende um conjunto de estados não acessíveis gerados pelo algoritmo $Rutt$. Em geral, na teoria de autômatos esses estados são considerados inúteis, pois ela considera que o processamento se

Figura 34 – LC $C_{1\to2}$ para a planta com caminho paralelo.



Fonte: Próprio Autor

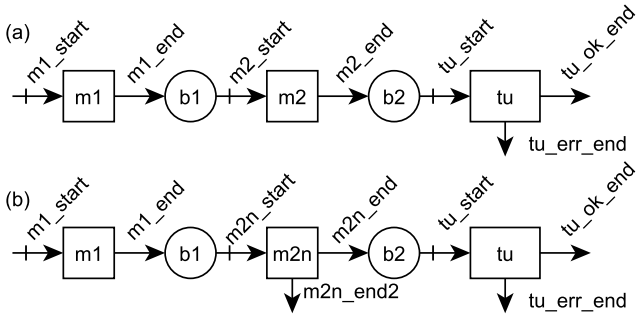
inicia sempre no estado inicial. No caso da RDS, quando a LC $C_{1\to2}$ toma o lugar de C_1, ela deve iniciar a partir de um estado equivalente ao que C_1 estava quando foi removida. Este estado pode representar uma especificação que não é mais tratada na nova LC e, então, $C_{1\to2}$ inicia a partir de um desses estados não acessíveis. Mas que dão acesso aos demais estados.

5.4 EXEMPLO DE UMA CÉLULA DE PRODUÇÃO COM DESCARTE

Essa seção apresenta um estudo de caso de RD, em um SMR, para demonstrar a substituição de uma das máquinas por outra com mais funcionalidades. Esse estudo de caso foi implementado numa célula didática e também numa simulação disponível em (PINTO; ROSSO; LEAL, 2018a). A Figura 35a mostra a configuração inicial do SMR, enquanto que a Figura 35b mostra a nova configuração desejada para o mesmo.

O SMR consiste em 2 máquinas $m1$ e $m2$, uma unidade de teste tu e 2 *buffers* $b1$ e $b2$

Figura 35 – (a) Uma célula de produção (b) A mesma célula de produção com a máquina $m2$ substituída por $m2n$.

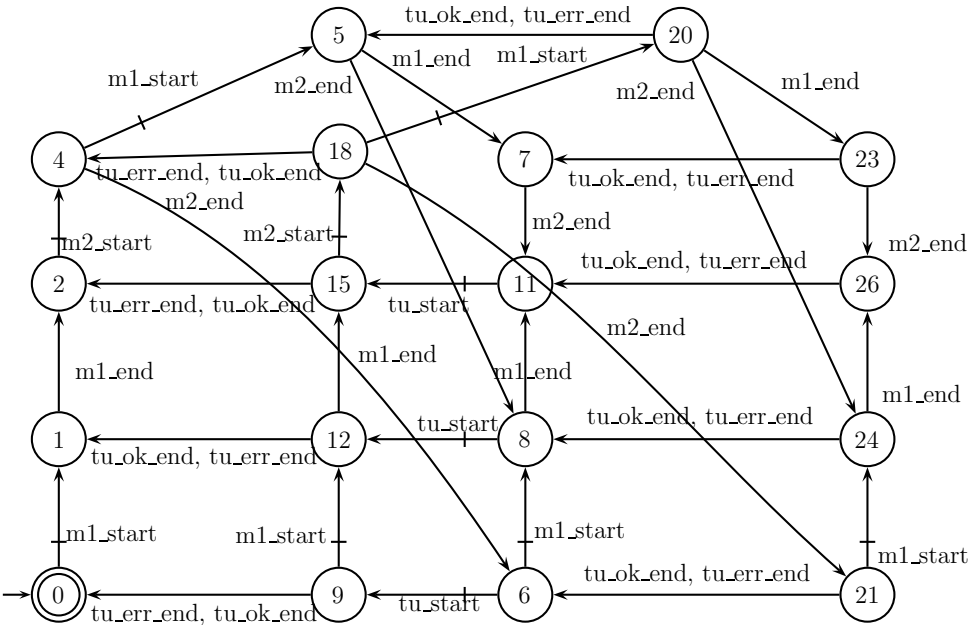


Fonte: Adaptado de (ZHONG; WONHAM, 1990).

com capacidade para uma peça cada. As peças fluem da entrada da máquina $m1$ e saem pela unidade de teste tu, a qual realiza um teste e aceita ou rejeita a peça. As máquina utilizam os *buffers* $b1$ e $b2$ como intermediários para transferir peças entre elas. Na Figura 35b, a máquina $m2$ foi substituída por $m2n$, a qual pode executar um tipo de teste e rejeitar uma peça antes que ela chegue até o final da célula.

A LC de ambas as configurações da planta foram geradas através da TCS. Para a planta da Figura 35a, a respectiva LC C_4 é representada pelo autômato na Figura 36. Ela contém 18 estados e 41 transições. Para a planta da Figura 35b, a respectiva LC C_5 é representada pelo autômato na Figura 37. Ela contém 18 estados e 47 transições.

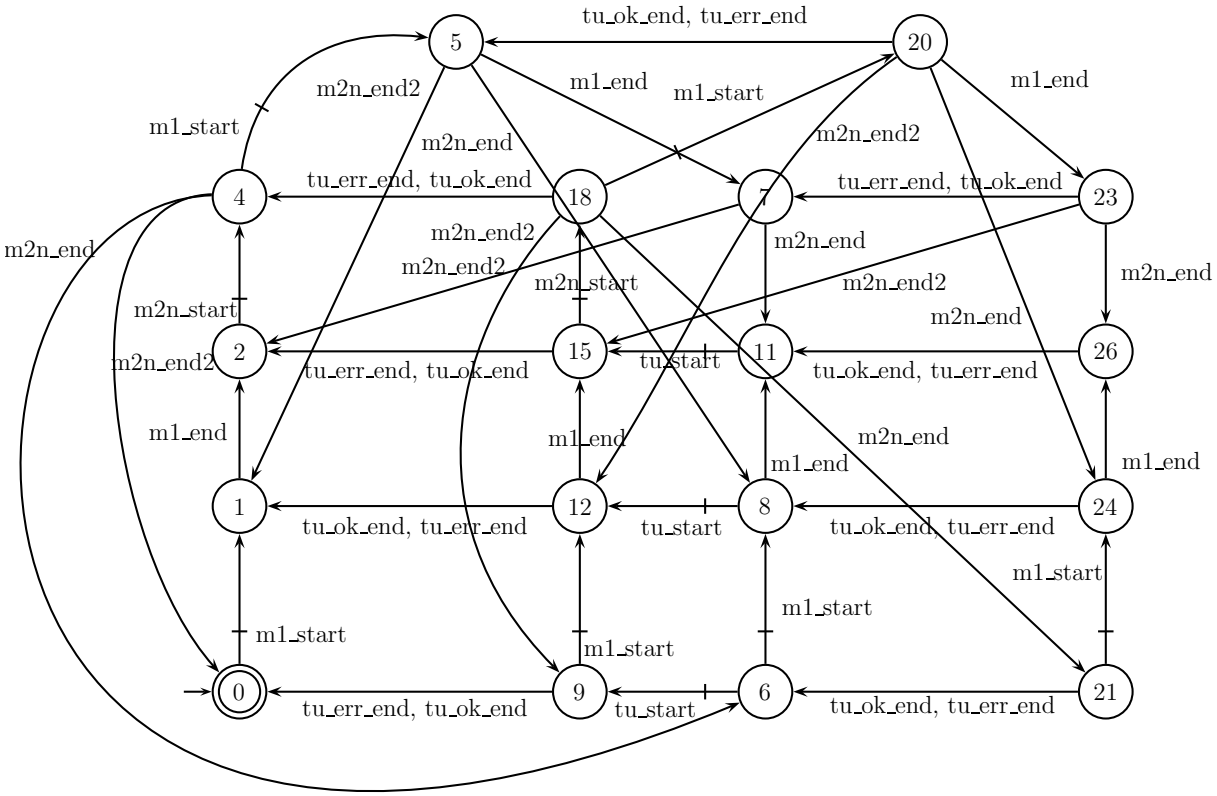
Figura 36 – LC C_4.



Fonte: Próprio Autor

Calculando $C_{4\to5} = Rutt(C_4, C_5)$, um autômato com 24 estados e 63 transições é

Figura 37 – LC C_5.



Fonte: Próprio Autor

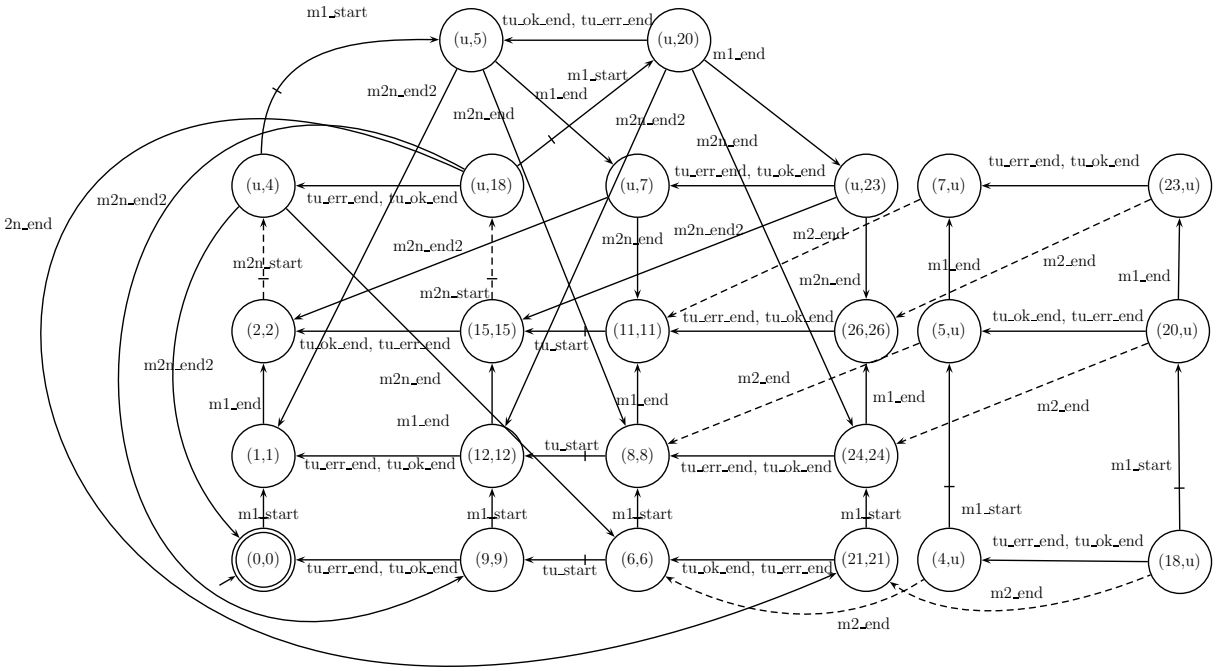
obtido. A Figura Figure 38 mostra esse autômato. Quando a RD é executada, o estado atual de C_4, denotado por q^c, é copiado e C_4 é substituído por $C_{4\to5}$. Entretanto, $C_{4\to5}$ deve iniciar de um estado equivalente, o qual será um destes: (q^c, u) ou (q^c, q^{C_2}). Por exemplo, se C_4 estava no estado 20, $C_{4\to5}$ deve inciar do estado $(20, u)$.

A região com estados na forma (q^{C_1}, u) ou (u, q^{C_2}) são regiões não seguras, as quais representam $m2$ ou $m2n$ em funcionamento, respectivamente. Quando $m2$ termina sua operação, o evento $m2_end$ é disparado, então, o autômato $C_{4\to5}$ entra nos estados equivalentes. O evento $m2_end$ está destacado em tracejada na Figura 38. O evento $m2n_start$ pode ser bloqueado enquanto o usuário faz a substituição de $m2$ por $m2n$. Este evento também está destacado em linha tracejada na mesma figura. Neste caso, o modelo $C_{4\to5}$ manterá o sistema operando nos estados equivalentes da forma (q^{C_1}, q^{C_2}). Quando o sistema executa o evento $m2_end$ no modelo $C_{4\to5}$, os estados (q^{C_1}, u) não são mais necessários.

5.5 CONSIDERAÇÕES DO CAPÍTULO

Neste capítulo foi apresentada a solução proposta de RD segura, denotada como RDS. Foram introduzidos os conceitos de estado equivalente, o algoritmo $Rutt$ e o processo prático de RD, o qual foi ilustrado na Figura 25. Nota-se que o usuário não precisa ter conhecimento

Figura 38 – Modelo $C_{4\to5}$



Fonte: Próprio Autor

detalhado do sistema para executar a RDS. O próprio método proposto identifica as diferenças entre a antiga e a nova LC, tomando as decisões para executar uma RD segura.

Foi apresentada uma comparação com o trabalho de Nahabedian et al. (2018) cujos resultados foram similares para o estudo de caso proposto. Ambos os métodos conduzem o SMR de um estado qualquer da LC inicial para a nova LC. Entretanto, o método da RDS apresenta maior eficiência na geração da LC que fará a condução de uma configuração para outra. Segundo Nahabedian et al. (2018), existem situações em que a RD não é possível e, então, o método desse não gera uma solução. A RDS considera que a RD é sempre possível, mesmo que o único estado equivalente seja o inicial (considerado sempre seguro).

Essa comparação evidenciou uma limitação da RDS que é: a substituição de uma máquina, por exemplo, a qual possui funcionalidade de retrabalho, com o objetivo de modificar o produto gerado pelo SMR, faz com o que foi reprovado pela máquina anterior seja retrabalhado pela nova, a qual tem um funcionalidade diferente. Isso pode ser aceitável ou não dependendo do SMR. O trabalho de Nahabedian et al. (2018) parece ter a mesma limitação.

Uma vantagem do método de Nahabedian et al. (2018) é que este possui uma formulação precisa. O método da RDS ainda não tem uma prova formal, o que pode ser colocado como trabalho futuro. Entretanto, os passos do algoritmo $Rutt$ foram elaborados levando em conta que as especificações das LCs C_1 e C_2 não podem ser modificadas. Assim, a parte 1 do algoritmo encontra caminhos que são iguais entre as duas LCs, gerando os estados equivalentes e transições entre eles. A parte 2 desvia para o estado inicial aqueles estados que representam o

estado inicial de C_2, o que significa que a partir deste há especificações de C_1 que deixaram de existir e que, em C_2, o destino é o estado inicial. A parte 3 acrescenta as especificações C_1 que devem ser tratadas uma última vez. Enquanto que a parte 4 acrescenta as especificações C_2 que devem ser seguidas a partir de então.

Umas das contribuições desta tese é que o método da RDS é sistemático, com uma sequência de passos padronizada que pode ser automatizada, como mostra o Capítulo 6. Para aplicação prática da RDS, o SMR precisa fornecer ferramentas adequadas a RD, isto o diferencia de um sistema de manufatura tradicional. Para isso, é necessário que os dispositivos permitam a modificação da LC em tempo de execução. Dispositivos aderentes a IEC 61499 implementam os comandos de reconfiguração da norma, logo, permitem a modificação da LC em tempo de execução. Entretanto, uma LC é modelada através de autômatos, enquanto a norma IEC 61499 é baseada em blocos de função (FB). Assim, o Capítulo 6 apresenta um método de conversão de autômatos em um FB e aproveita o estudo de caso da seção 5.4 para uma validação prática também da RDS.

6 AUTOMATIZANDO A RDS EM SISTEMAS ADERENTES À IEC 61499

Este capítulo tem dois principais objetivos, o de apresentar uma aplicação prática da RDS e também apresentar o método desenvolvido neste trabalho que é capaz de automatizar o processo da RDS em sistemas aderentes a IEC 61499.

As contribuições deste capítulo foram: (a) Desenvolvimento de uma arquitetura de rede de blocos de função para execução de uma LC; (b) desenvolvimento de um algoritmo capaz de gerar esta rede automaticamente; (c) Desenvolvimento de uma aplicação de reconfiguração dinâmica (ARD) que pode ser gerada automaticamente.

Em resumo, uma rede de FBs executa a lógica de controle inicial (C_1) do sistema. A ARD é responsável por alterar essa rede por uma com novas especificações (substituir por C_2). A ARD por sua vez é gerada automaticamente e é executada somente para a reconfiguração dinâmica. Após a substituição, a ARD é eliminada. A forma como a rede de FBs que executa uma LC e a ARD foram projetadas possibilita que ambas possam ser manipuladas automaticamente.

Para a automatização de um processo de RDS, faz-se necessária uma estrutura que possa ser controlada apenas pela LC. Neste capítulo, a estrutura é modelada através da norma IEC 61499 e executada pelo ambiente FBDK. Essa estrutura permite que uma modificação na LC seja transferida automaticamente para o SMR. Foi utilizada uma planta didática e uma simulação, ambas apresentaram os mesmos resultados. A implementação e a simulação podem ser encontrada em Pinto, Rosso e Leal (2018a).

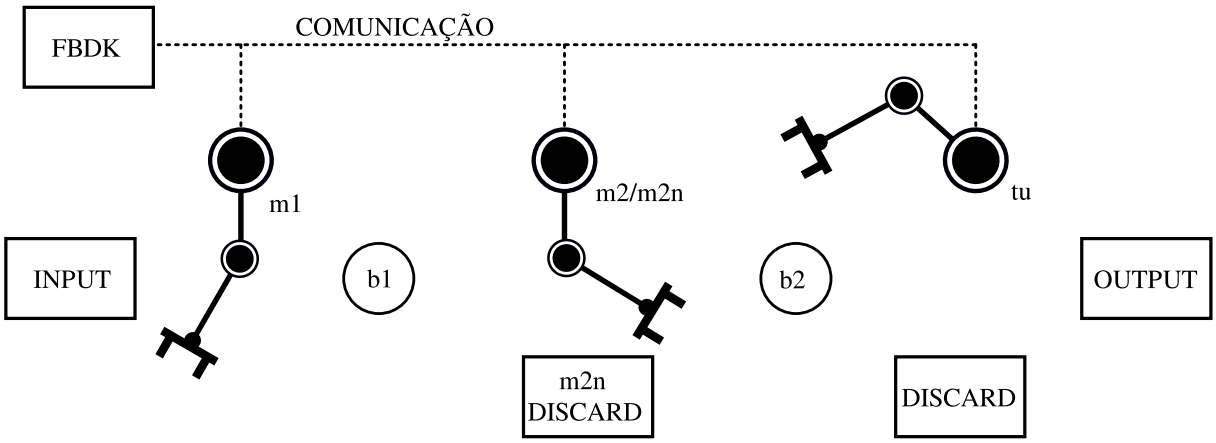
A Figura 39 ilustra a parte física da planta didática utilizada como estudo de caso neste capítulo. O modelo da planta com seus respectivos eventos já foi apresentado anteriormente na Figura 35 (Página 77). A planta consiste em três robôs que representam as três máquinas do SMR. O FBDK, que é executado em um computador, controla a planta através de SIFBs do tipo PUBLISHER e SUBSCRIBER. Um *gateway* foi desenvolvido para transportar informações entre os SIFBs, que utilizam protocolo TCP/IP, e os robôs, cuja interface de comunicação é serial.

6.1 ESTRUTURA ADERENTE A IEC 61499 PARA AUTOMATIZAÇÃO DA RDS

A implementação do processo de RDS num SMR aderente a IEC 61499 pode ser realizada com a estrutura ilustrada na Figura 40. Essa estrutura foi desenvolvida nesta tese para automatizar a RDS e demonstrar através da implementação do estudo de caso da célula de produção com descarte, a qual foi apresentado na Seção 5.4.

O processo da RDS consiste em substituir a LC atual por uma gerada pelo algoritmo

Figura 39 – Planta didática utilizada para este estudo de caso.



Fonte: Próprio Autor

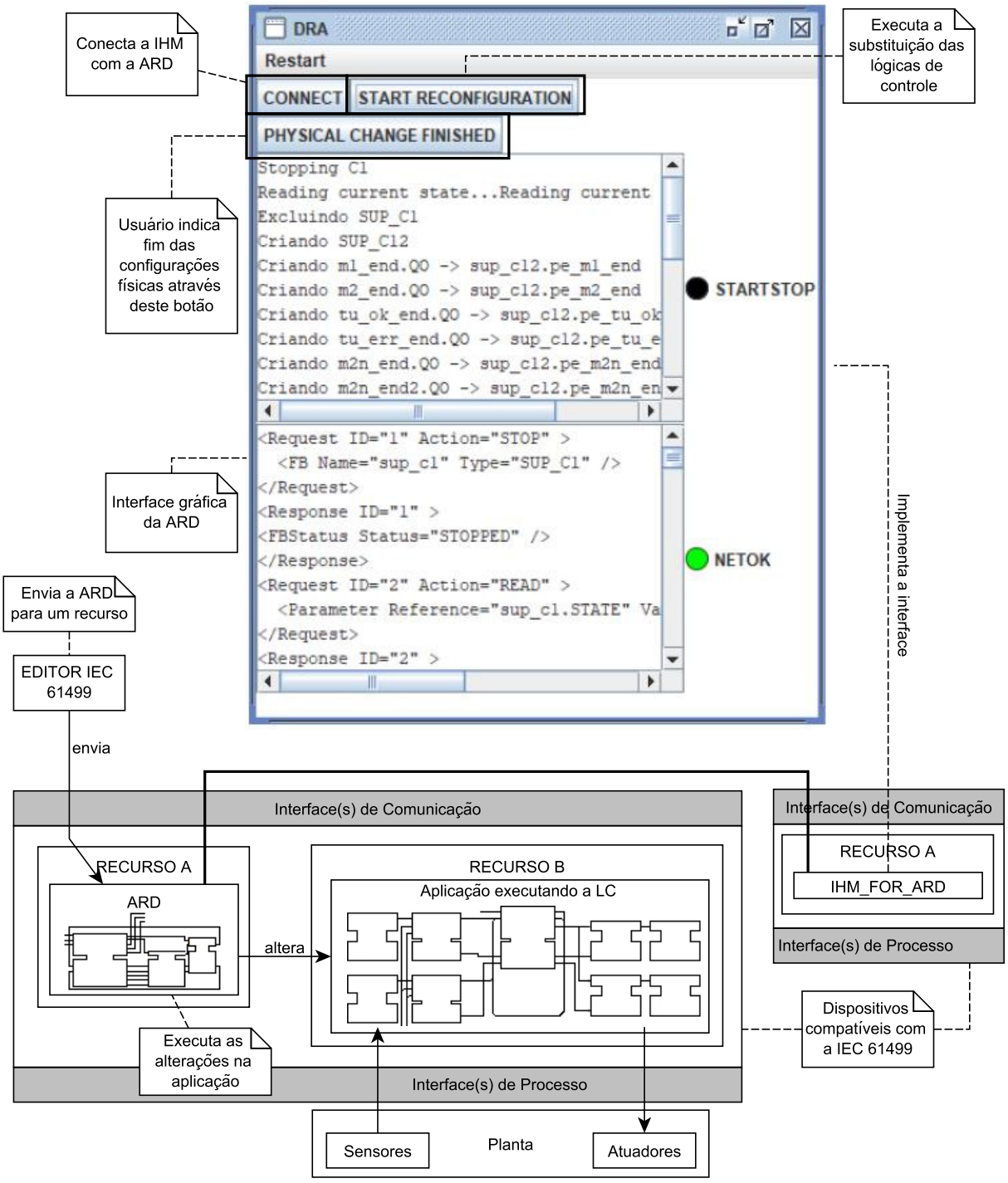
$Rutt$ e inicialmente com os novos eventos controláveis desabilitados, aguardar a entrada nos estados equivalentes, realizar modificações físicas e habilitar os novos eventos controláveis. Na Figura 40, a implementação realizada deste processo é ilustrada da seguinte forma: (a) Tem-se inicialmente um dispositivo executando a aplicação com a LC inicial; (b) Tendo a nova LC pronta, gera-se a nova aplicação e uma ARD, a qual executará a substituição, e a envia para o dispositivo, este pode ser carregado num RECURSO separado. Esse envio pode ser feito por uma ferramenta (EDITOR) compatível a IEC 61499. Para controle da ARD, carrega-se uma interface homem máquina (IHM) num dispositivo separado, neste caso um computador pessoal. Faz-se a conexão da IHM com a ARD através de seu botão *CONNECT*; (c) O usuário pressiona o botão *START RECONFIGURATION* para iniciar o processo de RDS. A ARD executa a substituição. Uma mensagem é mostrada ao usuário quando o sistema estiver nos estados equivalentes; (d) Após realizar as modificações físicas, o usuário pressiona o botão *PHYSICAL CHANGE FINISHED*. A ARD altera novamente a LC para habilitar os novos eventos controláveis.

Toda essa sequência de passos é automatizada, com exceção da FBN que implementa a LC. A parte desta FBN que acessa os sensores e atuadores deve ser implementada pelo usuário, normalmente são envolvidos blocos específicos do dispositivo como SIFBs e não padronizados pela norma. Esta parte é isolada em camadas que serão descritas na Seção 6.2. Entretanto, esse processo manual pode ser verificado pela ARD. É suficiente que a ARD execute uma busca por correspondências entre os eventos da LC e os blocos implementados pelo usuário.

6.2 ARQUITETURA DA FBN

Essa seção descreve a arquitetura de uma FBN desenvolvida nesta tese, a qual pode executar uma LC representada por um autômato. A Figura 41 mostra a estrutura dessa FBN. A arquitetura em camadas possibilita separar as partes que devem ser alteradas manualmente

Figura 40 – Exemplo de uma estrutura geral que implementa o método RDS num SMR aderente a IC 61499.



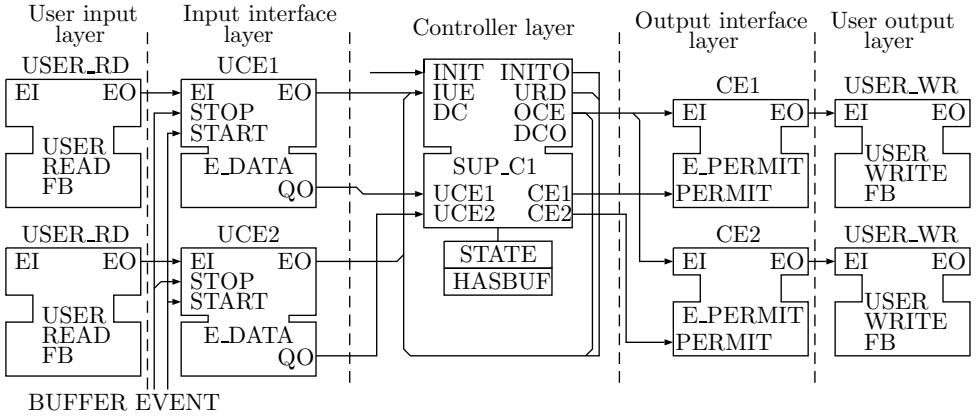
Fonte: Próprio Autor

pelo usuário das que são alteradas automaticamente pela ARD. Como o acesso as entradas e saídas dos dispositivos variam de um para outro, essas são isoladas nas camadas chamadas *User imput layer* e *User output layer*. Para que as entradas e saídas sejam acessíveis pela LC, que é implementada na camada *Controller layer*, as camadas *Input interface layer* e *Output interface layer* padronizam a comunicação com a camada central. Essa padronização permite a

substituição automatizada da LC (que se encontra no FB central SUP_C1) pela ARD.

O FB SUP_C1 implementa a LC, que é modelada por um autômato. Quando uma entrada é ativada, um evento é enviado para a camada de interface de entrada, e então, é enviada para SUP_C1. Ainda, quando um evento de saída é gerado, ele é enviado para a camada de interface de saída, e então, é enviado para a camada de saída do usuário.

Figura 41 – Uma representação genérica de uma FBN para implementar o modelo de um controlador.



Fonte: Próprio Autor

A camada de interface de entrada tem um mecanismo de armazenamento para guardar eventos enquanto é feita a substituição do FB SUP_C1. Essa camada evita a perda de eventos não controláveis como $m1_end$ e $m2_end$. O FB E_DATA passa a armazenar os eventos que chegam em EI quando STOP é disparado. O armazenamento é desligado quando START é disparado. Quando a rede está em operação normal, os eventos em EI são liberados para EO imediatamente. Os FBs da camada de interface de entrada possuem o mesmo nome das variáveis (eventos do SED) para as quais fazem o armazenamento. Isso torna mais fácil fazer a reconexão quando o FB SUP_C1 é substituído pela ARD. Por exemplo, se existe uma variável de entrada chamada UCE1, então existe um FB chamado UCE1 e basta fazer a conexão entre as variáveis UCE1.Q0 e SUP_C1.UCE1.

6.3 FB GERADO A PARTIR DO MODELO DE UM SUPERVISOR

Essa seção descreve como foram gerados os FBs que implementam as LCs C_4 e C_5 utilizadas neste capítulo. Essas LCs foram previamente utilizadas na Seção 5.4.

O FB chamado SUP_C1, mostrado na Figura 41, foi gerado automaticamente por um conjunto de algoritmos desenvolvidos nesta tese. Para simplificar o entendimento desse processo, a Figura 41 mostra a FBN de forma genérica, na qual a planta possui apenas dois eventos não controláveis (UCE1 e UCE2) e dois controláveis (CE1 e CE2). A interface é composta pelos seguintes eventos:

INIT: Para inicializar o FB;

INITO: Indica fim de inicialização do FB;

IUE: Ao disparar esse evento o bloco realiza a verificação se houve um evento não controlável na planta. Aqui são verificadas as variáveis UCE1 e UCE2, as quais são booleanos que representam o disparo de eventos da planta. Se não houve o disparo de nenhum evento não controlável, o FB dispara um evento controlável se houver no estado corrente;

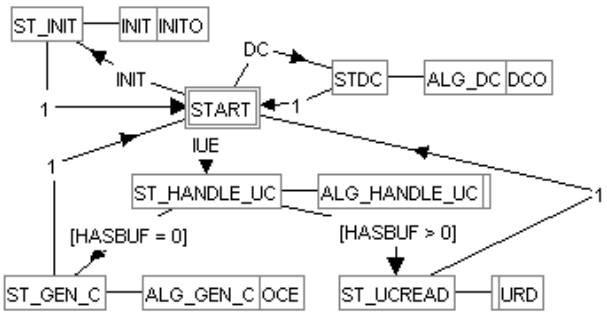
URD: Este evento indica o término do processamento iniciado por IUE. O evento é utilizado para reativar o FB.

DC: Ao disparar esse evento, são ativados os eventos controláveis que estavam desabilitados quando o bloco foi inserido na FBN. É utilizado no momento da RD;

DCO: Indica o fim do processamento iniciado por DC;

O ECC deste FB permanecem sempre igual mesmo para LCs diferentes, a Figura ilustra 42 o ECC. Ele executa ao disparar IUE, que aciona um algoritmo que verifica se um evento não controlável foi detectado nas entradas de variáveis booleanas, denotadas como UCEx. Os eventos não controláveis tem prioridade sobre os controláveis, assim, todos são tratados primeiro. Se não há mais eventos não controláveis para tratar, o ECC avança para o estado ST_GEN_C, o qual gera um evento controlável se existir no estado atual da LC.

Figura 42 – O ECC de um FB SUP_Cx.

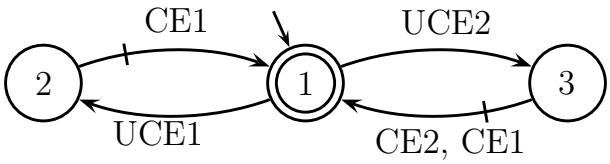


Fonte: Próprio Autor

O algoritmo ALG_HANDLE_UC é mostrado no Algoritmo 3. Para análise deste algoritmo, supõe-se que ele foi gerado para a LC C_1 ilustrada na Figura 43 que mostra esse autômato.

Para cada variável booleana UCEx existe uma variável interna do tipo inteiro peb_UCEx, utilizada para armazenar o disparo dos eventos da planta. Isso evita a perda de eventos caso mais de um chegue ao mesmo tempo. Observa-se no ECC que apenas um evento por vez é processado, portanto, outro evento que tenha chegado ao mesmo tempo será

Figura 43 – Exemplo de uma LC.



Fonte: Próprio Autor

Algoritmo 3: O algoritmo ALG_HANDLE_UC de um SUP_Cx FB.

```
if (UCE2.value) {
    peb_UCE2.value++; UCE2.value = false; HASBUF.value++;
}
if (UCE1.value) {
    peb_UCE1.value++; UCE1.value = false; HASBUF.value++;
}
if (HASBUF.value > 0) {
    if (STATE.value == 1) {
        if (peb_UCE1.value > 0) {
            peb_UCE1.value--;
            STATE.value = 2;
        } else if (peb_UCE2.value > 0) {
            peb_UCE2.value--;
            STATE.value = 3;
        }
    } else if (STATE.value == 3) {}
      else if (STATE.value == 2) {}
    HASBUF.value--;
}
```

processado no próximo ciclo. No algoritmo ALG_HANDLE_UC, o modelo do controlador é convertido para condicionais *if* para tratar cada estado do modelo. O estado muda dependendo do evento recebido. Neste algoritmo, tratam-se somente os eventos não controláveis.

Os eventos controláveis são tratados pelo algoritmo ALG_GEN_C do FB. Para o exemplo dado, esse algoritmo é mostrado no Algoritmo 4. Inicialmente, atribui-se *false* para as variáveis de saída do FB, as quais representam os eventos controláveis da planta. Em seguida, verifica-se em qual estado se encontra a LC. Se o estado possui apenas um evento controlável de saída, então esse evento é gerado para a planta, fazendo a sua respectiva variável de saída igual a $true$. Se o estado possui mais de um evento controlável de saída, então escolhe-se um aleatoriamente e o gera. A variável interna GLORAND auxilia na geração de um número aleatório para escolher um evento controlável, uma vez que a versão do FBDK utilizada neste trabalho não permite a importação de bibliotecas.

O FB SUP_C1 contém a variável interna DC para bloquear parte dos eventos

Algoritmo 4: O algoritmo ALG_GEN_C de um SUP_Cx FB.

```
pe_CE1.value = false;
pe_CE2.value = false;
if (STATE.value == 1) {} else if (STATE.value == 3) {
    int gen = GLORAND.value % 2;
    if (gen == 0) {
        pe_CE2.value = true;
        STATE.value = 1;
    }
    if (gen == 1) {
        pe_CE1.value = true;
        STATE.value = 1;
    }
} else if (STATE.value == 2) {
    pe_CE1.value = true;
    STATE.value = 1;
}
GLORAND.value++;
if (GLORAND.value > 100000) GLORAND.value = 0;
```

controláveis. Isto é útil para bloquear a execução de eventos que levam o sistema para estados não equivalentes, como, por exemplo, os novos eventos de uma máquina que foi adicionada ao sistema. Esse bloqueio permanece até que o usuário termine as alterações físicas.

Dois algoritmos da ferramenta de geração do FB – que gera a partir do autômato da LC – são responsáveis por gerar esses algoritmos internos do mesmo. Para cada LC, esses algoritmos internos são diferentes, pois dependem dos estados e eventos da LC.

Para gerar ALG_HANDLE_UC, foi desenvolvido o Algoritmo 5. Ele verifica todos os estados do autômato de entrada e gera todos os condicionais necessários para tratar cada estado e transição não controlável. O primeiro passo do algoritmo gera um condicional que incrementa o *buffer* do respectivo evento de planta que tenha ocorrido. Depois, são gerados os condicionais para verificar qual transição executar dependendo do estado corrente e do evento ocorrido.

Para gerar ALG_GEN_C, foi desenvolvido o Algoritmo 6. No passo inicial, ele gera os comandos para atribuir falso às variáveis de saída do FB, que correspondem aos eventos controláveis da planta. Depois, geram-se os condicionais para verificar qual o estado corrente da LC. Dentro de cada condicional, há um comando para atribuir verdadeiro a uma variável de saída, que representa um evento controlável da planta. Esse comando é o único dentro do condicional se houver apenas um evento controlável de saída do respectivo estado. Caso contrário, são gerados condicionais extras para escolher um evento aleatoriamente. Também é feita uma verificação para conferir se o evento gerado leva para um estado equivalente. Se o destino é um estado não equivalente, o condicional *if(!ISDC)* é acrescentado. ISDC é uma

Algoritmo 5: O algoritmo para gerar ALG_HANDLE_UC de um SUP_Cx FB.

Dados: G: Um autômato $(Q, \Sigma, \delta, q_s, M)$.
Resultado: Algoritmo para tratar eventos não controláveis.

```
1  para cada σ ∈ Σ^u (variável de entrada UCEx) faça
2      gerar("if(σ){ peb_σ++; HASBUF++; }");
3  gerar("if(HASBUF > 0){");
4  para cada estado s ∈ Q faça
5      se não é a primeira iteração então gerar(" else ") ;
6      gerar("if(STATE == s){");
7      para cada evento de saída σ ∈ Σ^u de s faça
8          se não é a primeira iteração então gerar(" else ") ;
9          gerar("if(peb_σ > 0){peb_σ = peb_σ − 1; STATE ← 8;}");
10     gerar("}");
11 gerar("HASBUF ← HASBUF - 1;}");
```

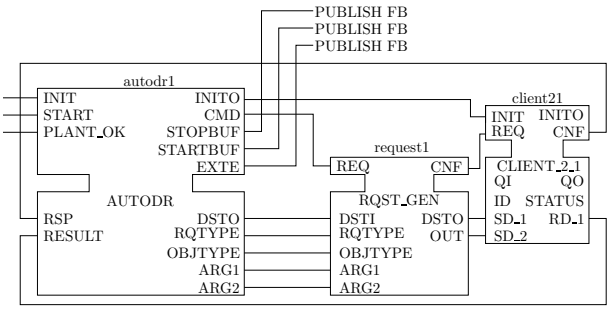
variável interna usada para liberar ou não a região crítica da LC. É utilizada quando o usuário precisa fazer alterações físicas na planta. Ao término destas, ele aciona o evento DC do FB que atribui falso a esta variável.

Um exemplo mais detalhado pode ser visto no Apêndice A, no qual é mostrado o FB gerado para executar a reconfiguração do SMR mostrado na Figura 35.

6.4 APLICAÇÃO DE RECONFIGURAÇÃO DINÂMICA

Os passos da RD podem ser executados por uma aplicação da IEC 61499. O FB AUTODR foi desenvolvido nesse trabalho para executar esses passos. A Figura 44 mostra o conceito principal da Aplicação de Reconfiguração Dinâmica (ARD), FBs de interação com o usuário foram omitidos. Esta aplicação contém uma interface gráfica para permitir que o usuário inicie a RD a qualquer momento.

Figura 44 – Representação geral da ARD.



Fonte: Próprio Autor

A ARD contém ainda um botão, na sua interface mostrada na Figura 45, para que o usuário indique que terminou as alterações físicas. Cada problema de RD é específico, então

Algoritmo 6: O algoritmo para gerar ALG_GEN_C de um SUP_Cx FB.

Dados: G: Um autômato $(Q, \Sigma, \delta, q_s, M)$.
Dados: Q_s: Conjunto dos estados equivalentes.
Resultado: Algoritmo para tratar eventos controláveis.

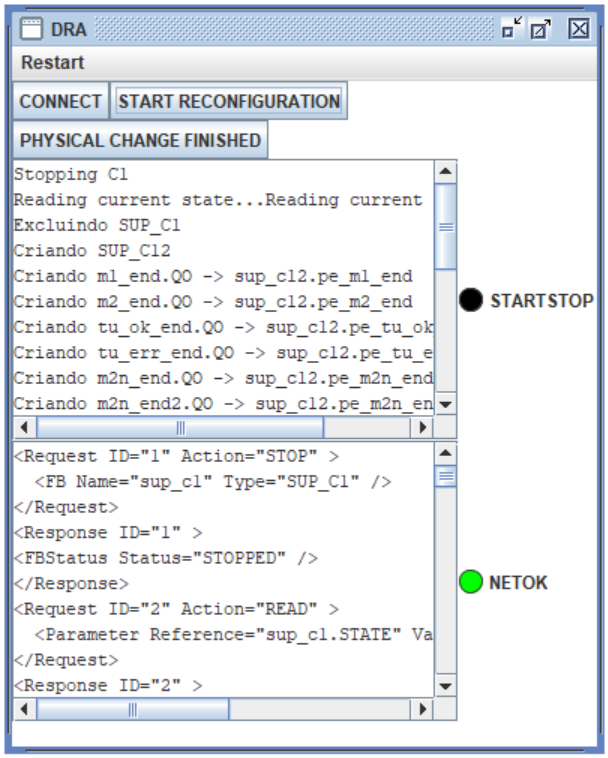
```
1  para cada σ ∈ Σ^c (variável de saída CEx) faça
2      gerar("σ = false");
3  para cada estado s ∈ Q faça
4      se não é a primeira iteração então gerar(" else ") ;
5      gerar("if(STATE == s){");
6      se s tem x > 1 eventos controlável de saída então
7          gerar("int gen = GLORAND.value % x;");
8          i ← 0;
9          para cada evento σ ∈ Σ^c que é saída de s faça
10             se δ(s, σ) ∈ Q_s então
11                 gerar("if (gen == i) if(!ISDC.value) σ.value = true; STATE.value
                       = δ(s, σ);")
12             senão
13                 gerar("if (gen == i) σ.value = true; STATE.value = δ(s, σ);")
14     senão
15         se s tem 1 evento controlável de saída então
16             se δ(s, σ) ∈ Q_s então
17                 gerar("if(!ISDC.value) σ.value = true; STATE.value = δ(s, σ);")
18             senão
19                 gerar(" σ.value = true; STATE.value = δ(s, σ);")
20     gerar("}");
21     gerar("GLORAND.value++; if(GLORAND.value > 100000)
           GLORAND.value = 0;");
```

uma nova ARD é gerada para cada processo de RD. Devido a estrutura de camadas adotada para a FBN, que é ilustrada na Figura 41, é possível gerar a ARD automaticamente. A ARD deve substituir o FB SUP_Cx, refazer as conexões com as camadas de entrada e saída e colocar o novo FB num estado equivalente ao anterior. Como o nome dos FBs nas camadas de interface são iguais aos nomes das variáveis do FB, um algoritmo pode determinar como conectar cada variável do FB.

O FB AUTODR envia os comandos de reconfiguração para o FB RQST_GEN, o qual gera o *xml*, com comandos de reconfiguração de acordo com a norma IEC 61499, e envia para CLIENT_2_1, estes dois FBs estão disponíveis no FBDK. Os comandos em *xml* são enviados para o dispositivo, que responde de volta para AUTODR. Então, o FB AUTODR executa o próximo comando de reconfiguração.

O ECC do FB AUTODR é mostrado na Figura 46. Os passos da RD são executados após o disparo do evento START, que é feito pelo usuário. No estado STOP, um evento é enviado

Figura 45 – Interface gráfica da ARD.



Fonte: Próprio Autor

para a interface de entrada da aplicação, causando a ativação dos *buffers* (Figura 41). Então, no estado REPLACE, o algoritmo ALG_REPLACE executa os seguintes passos: (a) interrompe a execução do FB SUP_Cx; (b) copia o seu estado; (c) exclui o bloco SUP_Cx; (d) cria o novo bloco SUP_Cx; (e) faz as conexões do FB com os FBs das camadas de entrada e saída. Uma máquina de estados implementada em ALG_REPLACE faz estas operações.

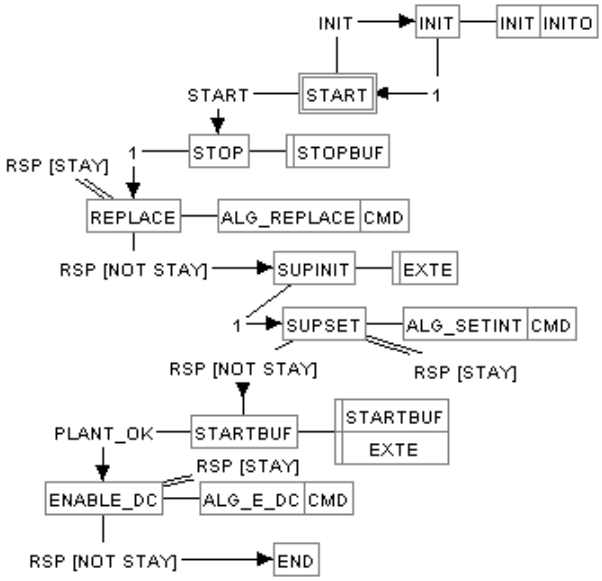
No último estado de ALG_REPLACE, a variável STAY passa a ser falsa, fazendo com que a próxima resposta de CLIENT_2_1 leve o ECC para o estado SUPINIT. No estado SUPINIT um evento é enviado para inicializar o novo FB, o estado desse FB é então colocado no mesmo estado que estava o anterior, antes de ser removido. No estado STARTBUF a camada de interface de entrada é reativada.

Finalmente, o novo SUP_Cx está em funcionamento. Caso alterações físicas sejam necessárias, o novo FB não executará os novos eventos controláveis. Quando essas alterações estiverem prontas, o usuário deve disparar o evento PLANT_OK, esse evento altera uma variável interna de SUP_Cx ativando os novos eventos controláveis.

6.5 GERANDO A ARD AUTOMATICAMENTE

A estrutura da aplicação de reconfiguração dinâmica (ARD) é padronizada e se mantém a mesma a cada processo de RD. Somente o FB AUTODR é diferente a cada RD. Essa seção

Figura 46 – O ECC do FB AUTODR.



Fonte: Próprio Autor

trata da geração deste FB. Internamente, o FB AUTODR é formado pelo ECC e os algoritmos:

ALG_REPLACE : Algoritmo que dispara os comandos para substituição do FB SUP_C1 por SUP_C12, os quais implementam as LCs C_1 e $C_{1\to2}$ respectivamente;

ALG_E_DC : Algoritmo que habilita os novos eventos controláveis;

ALG_SETINT : Algoritmo que coloca SUP_C12 no mesmo estado de SUP_C1 quando este é removido e também desabilita os novos eventos controláveis de SUP_C12;

ALG_READEVT : Algoritmo que lê o estado atual de SUP_C1.

O algoritmo gerador de ALG_REPLACE é mostrado no Algoritmo 7. Como parâmetros, o algoritmo recebe os nomes e tipos do FB a ser substituído e o novo FB, o qual implementa a LC $C_{1\to2}$. E também o conjunto de eventos controláveis (Σ^c) e não controláveis (Σ^u). Como resultado, o algoritmo retorna o algoritmo ALG_REPLACE em forma de *string* na variável T.

A linha 1 atribui à variável DSTO o nome do RECURSO no qual está a rede de FBs a ser modificada. A linha 2 inicializa a variável STAY que, enquanto tiver o valor *true*, o ECC do FB reexecuta o estado atual. A linha 4 cria o comando que para o FB atual. A linha 5 cria o comando que lê o estado atual do FB parado anteriormente. A linha 6 exclui o FB. A linha 7 cria o comando que instancia o novo FB e as linhas 8 a 10 criam as conexões padrões deste FB na rede.

Os laços nas linhas 11 e 14 criam os comandos para realizar as conexões com os FBs de entradas/saídas (E/S). Nota-se que os FBs de E/S possuem nomes iguais aos eventos em sí. Assim, basta conectar o evento ao bloco com nome correspondente.

A substituição do FB1 por FB12 se dá pela execução de uma sequência de comandos de reconfiguração da IEC 61499. O algoritmo ALG_REPLACE do FB AUTODR implementa internamente uma máquina de estados que executa um comando a cada chamada pelo ECC. O laço na linha 19 cria essa máquina de estados. A cada estado, as variáveis de saída são modificadas de acordo com o comando a ser executado. Esses dados são enviados para o FB RQST_GEN que cria uma *string* em *xml* para então enviar para execução no RECURSO de destino através do FB CLIENT_2_1.

Os demais algoritmos de AUTODR modificam-se somente em relação aos nomes e tipos dos FB que serão modificados. O algoritmo gerador destes modifica apenas estes pontos. Assim, um FB AUTODR completo é apresentado em *xml* no Apêndice B.

Algoritmo 7: O algoritmo para gerar ALG_REPLACE de um FB AUTODR.

Dados: FB1_nome, FB1_tipo: Nome e tipo do FB atualmente em execução.
Dados: FB12_nome, FB12_tipo: Nome e tipo do FB que irá substituir FB1.
Dados: Σ^c: conjunto de eventos controláveis de FB12.
Dados: Σ^u: conjunto eventos não controláveis de FB12.
Resultado: T: algoritmo ALG_REPLACE.

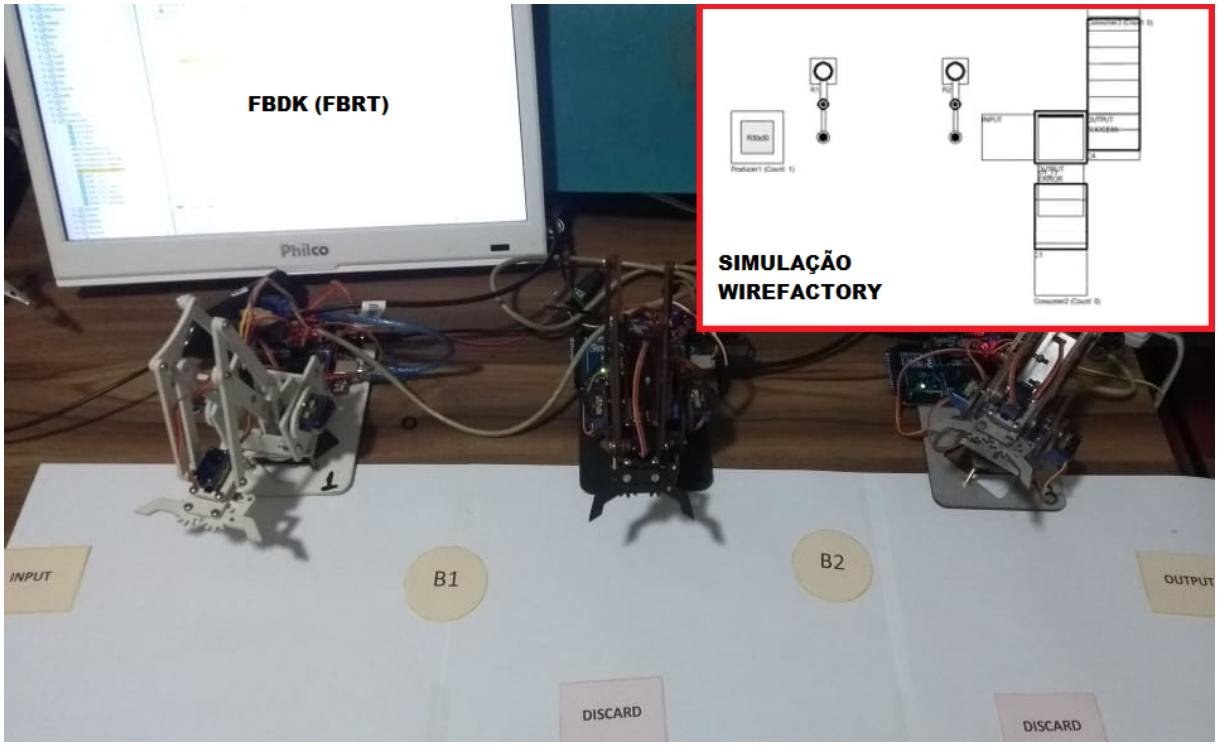
$T \leftarrow T +$ "DSTO.value = <RECURSO>;";
$T \leftarrow T +$ "STAY.value = true;";
$C \leftarrow$ lista de comandos ;
$C \leftarrow C + \{(\text{STOP}, \text{FB}, \text{FB1_nome}, \text{FB1_tipo})\}$;
$C \leftarrow C + \{(\text{READ, Parameter, FB1_nome.STATE, ' '})\}$;
$C \leftarrow C + \{(\text{DELETE}, \text{FB}, \text{FB1_nome}, \text{FB1_tipo})\}$;
$C \leftarrow C + \{(\text{CREATE}, \text{FB}, \text{FB12_nome}, \text{FB12_tipo})\}$;
$C \leftarrow C + \{(\text{CREATE}, \text{Connection}, \text{FB12_nome.INITO}, \text{FB12_nome.IUE})\}$;
$C \leftarrow C + \{(\text{CREATE}, \text{Connection}, \text{FB12_nome.URD}, \text{FB12_nome.IUE})\}$;
$C \leftarrow C + \{(\text{CREATE}, \text{Connection}, \text{FB12_nome.OCE}, \text{FB12_nome.IUE})\}$;
para cada $\sigma \in \Sigma^u$ **faça**
    $C \leftarrow C + \{(\text{CREATE}, \text{Connection}, \sigma\text{.QO}, \text{FB12_nome.}\sigma)\}$;
    $C \leftarrow C + \{(\text{CREATE}, \text{Connection}, \sigma\text{.EO}, \text{FB12_nome.IUE})\}$;
para cada $\sigma \in \Sigma^c$ **faça**
    $C \leftarrow C + \{(\text{CREATE}, \text{Connection}, \text{FB12_nome.}\sigma, \sigma\text{.PERMIT})\}$;
    $C \leftarrow C + \{(\text{CREATE}, \text{Connection}, \text{FB12_nome.OCE}, \sigma\text{.EI})\}$;
$C \leftarrow C + \{(\text{START}, \text{FB}, \text{FB12_nome}, \text{FB12_tipo})\}$;
$stateCount \leftarrow 0$;
para cada $c \in C$ **faça**
    se *não é a primeira iteração* **então**
        $T \leftarrow T +$ "else "
    $T \leftarrow T +$ "if(STATE.value == $stateCount$){";
    se *stateCount = 2* **então**
        $T \leftarrow T +$ "TARGET_STATE.value = estado recebido em RESULT;";
    $stateCount \leftarrow stateCount + 1$;
    $T \leftarrow T +$ "RQTYPE.value = $c.RQTYPE$;";
    $T \leftarrow T +$ "OBJTYPE.value = $c.OBJTYPE$;";
    $T \leftarrow T +$ "ARG1.value = $c.ARG1$;";
    $T \leftarrow T +$ "ARG2.value = $c.ARG2$;";
    $T \leftarrow T +$ "STATE.value = $stateCount$;";
    se *é o último comando* c *de* C **então**
        $T \leftarrow T +$ "STAY.value = false;";
    $T \leftarrow T +$ "}";

6.6 CONSIDERAÇÕES DO CAPÍTULO

Esse capítulo apresentou uma implementação que automatiza a RDS em SMR aderentes à IEC 61499. O estudo demonstrou a viabilidade dessa automatização e forneceu indícios para formalização de métodos que possibilitem a RD a nível de modelo, como por exemplo, a geração de um FB e uma ARD a partir da lógica de controle modelada por autômatos.

O estudo de caso apresentado como forma de validar o método proposto neste capítulo foi testado numa simulação através do simulador *WireFactory Simulator* (PINTO; ROSSO; LEAL, 2018b), desenvolvido durante esta tese, e também numa planta didática conforme ilustra a Figura 47. Um vídeo do funcionamento real pode ser obtido em (PINTO; ROSSO; LEAL, 2018a).

Figura 47 – Planta didática utilizada para este estudo de caso. No canto superior direito, a planta equivalente na simulação.



Fonte: Próprio Autor

Os testes foram executados com a LC C_4 sendo reconfigurada para $C_{4\to5}$. Então, os respectivos FBs SUP_C4 e SUP_C45 foram gerados. O ambiente de execução FBRT foi utilizado para executar a FBN, esse ambiente possui recursos para execução da RD fornecidas pelo FB DM_KRNL (HOLOBLOC Inc., 2018a).

Com o sistema executando SUP_C4, a ARD foi carregada e iniciada. Um botão na interface permite iniciar o processo de RD. A ARD substituiu o FB SUP_C4 por SUP_C45 e

aguardou o sistema atingir os estados equivalentes para informar que as mudanças físicas podem ser feitas. Após a substituição física de $m2$ por $m2n$, foi pressionado um botão na interface da ARD para indicar o fim das alterações físicas. A ARD então reativou os eventos controláveis dessa máquina. Mesmo durante a alteração, as demais máquinas do sistema continuaram executando até onde foi possível. Esta RD foi repetida várias vezes em diferentes estados de execução de SUP_C4. Os testes deste SMR demonstraram que é possível fazer as alterações a qualquer momento. O FB SUP_C45, cujo modelo foi gerado pela solução proposta nesse trabalho, mostrou ser capaz de controlar o sistema corretamente a partir de qualquer estado ou situação.

A implementação de um SMR, com o método RDS automatizado, apresenta a vantagem de permitir que o projetista atue com alto nível de abstração. Ele precisa atuar em 3 etapas: (a) elaboração da nova LC; (b) identificação e elaboração dos FBs das entradas e saídas para a FBN; (c) Alterações físicas do SMR. Logo, é eliminado do processo a necessidade de projetar ARD manualmente, pois a RDS fornece os passos para a RD.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como objetivo desenvolver um método de reconfiguração dinâmica segura (RDS) e aplicá-lo em sistemas aderentes à IEC 61499, para essa aplicação também foi desenvolvido um método para automatizar o processo de RD. Basicamente, o problema consiste em controlar corretamente o sistema de manufatura reconfigurável (SMR) durante o processo de reconfiguração dinâmica (RD) em tempo de execução, o qual deve remover as especificações antigas e inserir as novas adequadamente, além de possibilitar o tratamento de alterações físicas, como substituição de máquinas. Após introduzir a IEC 61499 e a TCS, foi proposto neste trabalho o método de reconfiguração dinâmica segura RDS e a sua aplicação em sistemas aderentes a IEC 61499. O método combina a lógica de controle inicial e a nova gerando uma solução para o problema.

Através da revisão da literatura sobre a RD em sistemas aderentes a IEC 61499, foi verificado que a maioria dos métodos de RD dependem de algum tipo de planejamento manual para o processo e que não há uma verificação formal do mesmo. O método proposto nesta tese tem como objetivo ser correto por construção e eliminar o trabalho manual de planejamento. Desta forma, foi possível automatizar o processo de RD. Outra lacuna identificada é a alteração pelo processo de RD da parte física do sistema de forma não prevista no projeto da planta.

Para alcançar o objetivo de automatização do método, uma implementação aderente a IEC 61499 foi desenvolvida. Esta contribuiu para demonstrar a viabilidade do método, sua aplicação prática e demonstrar o método de geração automática a partir de modelos desenvolvidos com autômatos. Em resumo, o projetista pode trabalhar em nível de modelo, enquanto as camadas até a execução são bem definidas a fim de permitir a automatização. A reconfiguração dinâmica segura (RDS), apresentada nesse trabalho, mostrou que é possível modificar um sistema de automação e controle em tempo de execução preservando a consistência do mesmo de forma automatizada e projetando a nível de modelo.

A RDS baseia-se no algoritmo *Rutt* (Algoritmo 2 – página 66) para encontrar uma LC adequada para a RD. A LC obtida pode substituir a atualmente em execução a qualquer momento, sendo suficiente a cópia do estado da antiga LC para a nova. Há sempre ao menos um estado equivalente entre duas LCs, que é o estado inicial. A RDS consegue localizar outros estados equivalentes entre duas LCs, através de caminhos equivalentes entre os autômatos que representam essas LCs. Quando a nova LC remove antigas especificações, deixam de existir estados equivalentes da antiga LC para a nova. Nesse caso, são criados estados não acessíveis na nova LC que tratam essas especificações, removendo-as. Não foram encontrados na literatura trabalhos que utilizassem estados não acessíveis para este propósito. Além disso, a LC gerada pelo algoritmo *Rutt* responde as questões anteriormente levantadas nos trabalhos relacionados

(Capítulo 4) como, por exemplo, qual o momento correto de substituir a LC? Qual o momento correto de modificar a parte física do sistema? E também, como isolar as regiões críticas para possibilitar as modificações?

A RDS proposta se diferencia das demais abordagens de RD por fornecer um processo de RD que pode ser automatizado. O projetista fornece somente a LC atualmente em execução e a nova LC desejada. Os passos da reconfiguração foram definidos no Capítulo 5 e exemplificados na aplicação prática no Capítulo 6, no qual foi demonstrada a capacidade de realizar alterações físicas no sistema, que é uma lacuna identificada nos trabalhos relacionados.

A TCS foi utilizada para gerar a LC menos restritiva em alguns dos estudos de caso. Com essa abordagem, encontra-se o maior número possível de estados equivalentes, o que significa que o sistema pode realizar mais tarefas durante a RD.

A TCS fornece ainda o conceito de eventos controláveis e não controláveis. Através dos eventos controláveis tornou-se possível a RD da parte física do sistema, pois torna-se possível impedir a evolução de parte do sistema momentaneamente em pontos chaves a fim de possibilitar modificações efetuadas por pessoas. O método de RDS encontra automaticamente esses pontos, que são os novos eventos controláveis, e impede o sistema de passar por eles. Deve-se indicar manualmente quando tal passagem é permitida, o que foi feito através de um botão na ARD apresentada no Capítulo 6.

Neste trabalho optou-se por utilizar um dispositivo aderentes a IEC 61499 para implementar a RDS proposta, a qual é independente de tecnologia. A IEC 61499 favorece o uso da RDS por ser uma norma que incorpora recursos para RD e é uma padronização internacional.

7.1 LIMITAÇÕES E TRABALHOS FUTUROS

Essa seção trata das limitações encontradas no método RDS proposto. A primeira delas trata-se do caso particular no qual uma máquina coloca uma peça dentro de outra máquina. Nesse caso não há um *buffer* entre as máquinas. Logo, se uma máquina será modificada na linha de produção, deve-se observar se não existe outra máquina que coloque peças dentro dessa. Supondo uma linha na qual um robô coloca peças dentro de uma máquina CNC, a RDS garante que a CNC não será acionada quando for substituída, mas não pode garantir o não acionamento do robô. Como visto no Capítulo 5, a RDS permite executar o máximo de operações durante uma RD. Assim, um trabalho futuro poderia propor uma forma de modelar essa situação e incorporar uma solução no algoritmo *Rutt*.

Este trabalho modela SEDs através de autômatos finitos determinísticos (AFDs). Porém o uso de autômatos temporizados também é possível. Por exemplo, supondo que um robô deve retirar uma peça de uma solução ácida após ela ter permanecido lá por 20 minutos, uma RD com intenção de substituir o robô deve levar isso em consideração. Um AFD não considera o tempo nos estados do sistema, então não poderia informar sobre o problema dos

20 minutos. Nessa situação, o sistema permaneceria em execução até uma peça ser colocada na solução ácida e, então, aguardaria o término da substituição do robô. Um trabalho futuro poderia propor uma solução na qual a RDS informe sobre os 20 minutos ou impeça a peça de ser colocada na solução ácida, marcando essa situação como um estado não equivalente. Assim, o algoritmo *Rutt* poderia avaliar os estados críticos em relação ao tempo.

Como trabalho futuro, uma prova formal para a comprovação do algoritmo *Rutt* poderia ser desenvolvida. Os passos do algoritmo *Rutt* foram desenvolvidos com base na teoria de autômatos. Foi suposto que existem similaridades entre LCs quaisquer. Quando uma LC é apenas uma nova versão de uma anterior, essas similaridades aumentam. Os testes realizados demonstram essa característica.

Em sistemas de manufatura distribuídos e mais complexos, mais de uma LC pode ser empregada para o controle do mesmo. Portanto, um trabalho futuro poderá explorar os aspectos de RD quando mais de uma LC é utilizada. Existe também uma abordagem modular local da Teoria de Controle Supervisório que é mais adequada para se obter a LCs modulares (QUEIROZ; CURY, 2000). Como uma das principais propostas da IEC 61499 é a utilização em sistemas distribuídos, um trabalho futuro neste sentido poderia utilizar os recursos de distribuição da IEC 61499. Na mesma linha existe ainda o autômato de Buchi, o qual também é apropriado para uso em sistemas a eventos discretos, como mostra o trabalho de (RAMADGE, 1989). A principal característica do autômato de Buchi é a capacidade de reconhecimento de cadeias (ou sequência de eventos) infinitas.

REFERÊNCIAS

ALMEIDA, E. E.; LUNTZ, J. E.; TILBURY, D. M. Event-condition-action systems for reconfigurable logic control. **IEEE Transactions on Automation Science and Engineering**, v. 4, n. 2, p. 167–181, April 2007. ISSN 1545-5955.

ALSAFI, Y.; VYATKIN, V. Ontology-based reconfiguration agent for intelligent mechatronic systems in flexible manufacturing. **Robotics and Computer-Integrated Manufacturing**, v. 26, n. 4, p. 381 – 391, 2010. ISSN 0736-5845.

ATMOJO, U. D. et al. Distributed control architecture for dynamic reconfiguration: Flexible assembly line case study. In: **2018 IEEE Industrial Cyber-Physical Systems (ICPS)**. [S.l.: s.n.], 2018. p. 690–695.

ATMOJO, U. D.; SALCIC, Z.; WANG, K. I. Dynamic reconfiguration and adaptation of manufacturing systems using sosj framework. **IEEE Transactions on Industrial Informatics**, v. 14, n. 6, p. 2353–2363, June 2018. ISSN 1551-3203.

BLECH, J. O. et al. A comparison of formal verification approaches for IEC 61499. In: **2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)**. [S.l.: s.n.], 2016. p. 1–4.

CASSANDRAS, C. G.; LAFORTUNE, S. **Introduction to Discrete Event Systems**. 2nd. ed. [S.l.]: Springer US, 2008. ISBN 978-0-387-33332-8.

CENGIC, G.; AKESSON, K. On formal analysis of IEC 61499 applications, part B: Execution semantics. **Industrial Informatics, IEEE Transactions on**, v. 6, n. 2, p. 145–154, May 2010. ISSN 1551-3203.

CHOUINARD, J.; BRENNAN, R. W. Software for next generation automation and control. In: **IEEE International Conference on Industrial Informatics**. [S.l.: s.n.], 2006. p. 886–891. ISBN 0-7803-9700-2.

DAI, W. et al. Bridging service-oriented architecture and iec 61499 for flexibility and interoperability. **Industrial Informatics, IEEE Transactions on**, v. 11, n. 3, p. 771–781, June 2015. ISSN 1551-3203.

DUBININ, V.; VYATKIN, V. Semantics-robust design patterns for IEC 61499. **Industrial Informatics, IEEE Transactions on**, v. 8, n. 2, p. 279–290, May 2012. ISSN 1551-3203.

DUBININ, V.; VYATKIN, V.; SHALYTO, A. Formal modeling and verification of IEC 61499 function blocks on the basis of transition systems. In: **2016 International Siberian Conference on Control and Communications (SIBCON)**. [S.l.: s.n.], 2016. p. 1–4.

Eclipse 4DIAC. **Framework for Distributed Industrial Automation and Control**. <http://www.fordiac.org/>: [s.n.], 2019.

FENG, L.; WONHAM, W. M. Tct: A computation tool for supervisory control synthesis. In: **Discrete Event Systems, 2006 IEEE 8th International Workshop on**. [S.l.: s.n.], 2006. p. 388–389.

FERRARINI, L.; VEBER, C. Implementation approaches for the execution model of iec 61499 applications. In: **Industrial Informatics, 2004. INDIN '04. 2004 2nd IEEE International Conference on**. [S.l.: s.n.], 2004. p. 612–617.

FREY, G.; HUSSAIN, T. Modeling techniques for distributed control systems based on the IEC 61499 standard - current approaches and open problems. In: **Discrete Event Systems, 2006 8th International Workshop on**. [S.l.: s.n.], 2006. p. 176–181.

Guellouz, S. et al. Reconfigurable function blocks: Extension to the standard iec 61499. In: **2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)**. [S.l.: s.n.], 2016. p. 1–8. ISSN 2161-5330.

HAGGE, N.; WAGNER, B. A new function block modeling language based on petri nets for automatic code generation. **IEEE Transactions on Industrial Informatics**, v. 1, n. 4, p. 226–237, Nov 2005. ISSN 1551-3203.

HALL, K.; STARON, R.; ZOITL, A. Challenges to industry adoption of the iec 61499 standard on event-based function blocks. In: **Industrial Informatics, 2007 5th IEEE International Conference on**. Vienna: IEEE, 2007. v. 2, p. 823 –828. ISSN 1935-4576.

HOLOBLOC Inc. **FUNCTION_BLOCK DM_KRNL**. 2018. [Online]. Available: <http://www.holobloc.com/doc/fb/rt/DM_KRNL.htm>.

HOLOBLOC Inc. **HOLOBLOC.com - FBDK - Function Block Development Kit**. 2018. [Online]. Available: <http://www.holobloc.com/>.

HOPCROFT, J. E. et al. **Introduction to Automata Theory, Languages and Computability**. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201441241.

IEC 61131-3. **International Electrotechnical Commission: Programmable controllers - Part 3: Programming languages**. [S.l.], 2003.

IEC 61499-1. **European Committee for Electrotechnical Standardization - BS EN 61499-1: Function Blocks - Part 1: Architecture**. [S.l.], 2012.

IEC 61499-2. **European Committee for Electrotechnical Standardization - BS EN 61499-2: Function Blocks - Part 2: Software tool requirements**. [S.l.], 2012.

IEC 61499-4. **European Committee for Electrotechnical Standardization - BS EN 61499-4: Function Blocks - Part 4: Rules for compliance profiles**. [S.l.], 2013.

KRAMER, J.; MAGEE, J. Dynamic configuration for distributed systems. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 11, n. 4, p. 424–436, abr. 1985. ISSN 0098-5589.

KUO, M. et al. Determining the worst-case reaction time of iec 61499 function blocks. In: **Industrial Informatics (INDIN), 2010 8th IEEE International Conference on**. Osaka: IEEE, 2010. p. 1104–1109.

LEAL, A. B.; CURY, J. E. Modular supervision of hybrid systems: a des approach. **IFAC Proceedings Volumes**, v. 37, n. 18, p. 165 – 170, 2004. ISSN 1474-6670. 7th International Workshop on Discrete Event Systems (WODES'04), Reims, France, September 22-24, 2004. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1474667017307401>.

LEWIS, R. **Modelling Control Systems Using IEC 61499: Applying function blocks to distributed systems**. 1th. ed. [S.l.]: Institution of Engineering and Technology, 2008.

LOYTEC. **LDALI-PLC4: Programmable DALI Controller**. [S.l.], 2019.

MACKTOOBIAN, M.; WONHAM, W. M. Automatic reconfiguration of untimed discrete-event systems. In: **2017 14th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)**. [S.l.: s.n.], 2017. p. 1–6.

NAHABEDIAN, L. et al. Dynamic update of discrete event controllers. **IEEE Transactions on Software Engineering**, p. 1–1, 2018. ISSN 0098-5589.

OLSEN, S. et al. Contingencies-based reconfiguration of distributed factory automation. **Robotics and Computer-Integrated Manufacturing**, v. 21, n. 4-5, p. 379 – 390, 2005. ISSN 0736-5845. 14th International Conference on Flexible Automation and Intelligent Manufacturing.

PANG, C.; VYATKIN, V. Towards formal verification of iec61499: modelling of data and algorithms in nces. In: **Industrial Informatics, 2007 5th IEEE International Conference on**. [S.l.: s.n.], 2007. v. 2, p. 879–884. ISSN 1935-4576.

PANG, C.; VYATKIN, V. Automatic model generation of IEC 61499 function block using net condition/event systems. In: **Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on**. [S.l.: s.n.], 2008. p. 1133–1138. ISSN 1935-4576.

PINHEIRO, L. P. et al. Nadzoru: A software tool for supervisory control of discrete event systems. **IFAC-PapersOnLine**, v. 48, n. 7, p. 182 – 187, 2015. ISSN 2405-8963.

PINTO, L. I.; LEAL, A. B.; ROSSO, R. S. U. Safe dynamic reconfiguration through supervisory control in IEC 61499 compliant systems. In: **2017 IEEE 15th International Conference on Industrial Informatics (INDIN)**. [S.l.: s.n.], 2017. p. 753–758.

PINTO, L. I.; ROSSO, R. S. U.; LEAL, A. B. **Rutt Algorithm**. 2018. [Online]. Available: <https://github.com/leandroisrael/rutt>.

PINTO, L. I.; ROSSO, R. S. U.; LEAL, A. B. **WireFactory Simulator**. 2018. [Online]. Available: <https://sourceforge.net/projects/wirefactory/>.

PINTO, L. I. et al. Icaru-fb: An IEC 61499 compliant multiplatform software infrastructure. **IEEE Trans. on Industrial Informatics**, v. 12, n. 3, p. 1074–1083, June 2016. ISSN 1551-3203.

PRENZEL, L.; PROVOST, J. Dynamic software updating of iec 61499 implementation using erlang runtime system. **IFAC-PapersOnLine**, v. 50, n. 1, p. 12416 – 12421, 2017. ISSN 2405-8963. 20th IFAC World Congress. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2405896317332950>.

QUEIROZ, M. H. de; CURY, J. E. R. Modular supervisory control of large scale discrete event systems. In: ____. **Discrete Event Systems: Analysis and Control**. In: R. Boel; G. Stremersch. (Org.). Discrete event systems: analysis and control. 1ed. Massachusetts: Kluwer Academic Publishers: [s.n.], 2000. v. 1, p. 103–110. ISBN 978-1-4615-4493-7.

RAMADGE, P. J.; WONHAM, W. M. Supervision of discrete event processes. In: **1982 21st IEEE Conference on Decision and Control**. [S.l.: s.n.], 1982. p. 1228–1229.

RAMADGE, P. J. G. Some tractable supervisory control problems for discrete-event systems modeled by Buchi automata. **IEEE Transactions on Automatic Control**, v. 34, n. 1, p. 10–19, Jan 1989.

RAUSCH, M.; HANISCH, H.-M. Net condition/event systems with multiple condition outputs. In: **Emerging Technologies and Factory Automation, 1995. ETFA '95, Proceedings., 1995 INRIA/IEEE Symposium on**. [S.l.: s.n.], 1995. v. 1, p. 592–600 vol.1.

Rockwell Automation. **ISaGRAf**. [S.l.], 2019.

RUDIE, K. The integrated discrete-event systems tool. In: **Discrete Event Systems, 2006 IEEE 8th International Workshop on**. [S.l.: s.n.], 2006. p. 394–395.

SAMPATH, R. et al. Control reconfiguration of discrete event systems with dynamic control specifications. **IEEE Transactions on Automation Science and Engineering**, v. 5, n. 1, p. 84–100, Jan 2008. ISSN 1545-5955.

Schneider Electric. **nxtRT61499F - run time systems for distributed control engineering**. 2015.

SEIFZADEH, H.; ABOLHASSANI, H.; MOSHKENANI, M. S. A survey of dynamic software updating. **Journal of Software: Evolution and Process**, v. 25, n. 5, p. 535–568, 2013. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1556>.

SREENIVAS, R.; KROGH, B. On condition/event systems with discrete state realizations. **Discrete Event Dynamic Systems**, Kluwer Academic Publishers, v. 1, n. 2, p. 209–236, 1991. ISSN 0924-6703. Disponível em: <http://dx.doi.org/10.1007/BF01805563>.

STRASSER, T. et al. Framework for distributed industrial automation and control (4diac). In: **Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on**. [S.l.: s.n.], 2008. p. 283 –288. ISSN 1935-4576.

STRASSER, T. et al. Standardized dynamic reconfiguration of control applications in industrial systems. **International Journal of Applied Industrial Engineering (IJAIE)**, v. 2, p. 57–73, 2014. ISSN 2155-4153, 2155-4161.

STRASSER, T. et al. Design and execution issues in iec 61499 distributed automation and control systems. **Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on**, v. 41, n. 1, p. 41–51, 2011. ISSN 1094-6977.

SüNDER, C.; VYATKIN, V.; ZOITL, A. Formal verification of downtimeless system evolution in embedded automation controllers. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 12, n. 1, p. 17:1–17:17, jan. 2013. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/2406336.2406353>.

SUNDER, C. et al. Usability and interoperability of iec 61499 based distributed automation systems. In: **Industrial Informatics, 2006 IEEE International Conference on**. [S.l.: s.n.], 2006. p. 31–37.

TEIXEIRA, M.; CURY, J. E. R.; QUEIROZ, M. H. de. Local modular supervisory control of des with distinguishers. In: **ETFA2011**. [S.l.: s.n.], 2011. p. 1–8. ISSN 1946-0740.

VYATKIN, V. The IEC 61499 standard and its semantics. **Industrial Electronics Magazine, IEEE**, v. 3, n. 4, p. 40–48, 2009. ISSN 1932-4529.

VYATKIN, V. IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. **Industrial Informatics, IEEE Transactions on**, v. 7, n. 4, p. 768–781, Nov 2011. ISSN 1551-3203.

VYATKIN, V. **IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design**. 2th. ed. [S.l.]: International Society of Automation, 2012. ISBN 978-1-936007-93-6.

VYATKIN, V.; HANISCH, H.-M. Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems. In: **Emerging Technologies and Factory Automation, 2001. Proceedings. 2001 8th IEEE International Conference on**. [S.l.: s.n.], 2001. v. 2, p. 113–118 vol.2.

WURMUS, H.; WAGNER, B. IEC 61499 konforme beschreibung verteilter steuerungen mit petri-netzen. **Proc. Fachtagung Verteilte Automatisierung**, Kluwer Academic Publishers, 2000. Disponível em: <http://dx.doi.org/10.1007/BF01805563>.

YOONG, L. H.; ROOP, P.; SALCIC, Z. Efficient implementation of iec 61499 function blocks. In: **Industrial Technology, 2009. ICIT 2009. IEEE International Conference on**. [S.l.: s.n.], 2009. p. 1 –6.

ZHANG, W.; DIEDRICH, C.; HALANG, W. Module and integration verifications for function block-based safety-related system development. In: **Industrial Informatics, 2004. INDIN '04. 2004 2nd IEEE International Conference on**. [S.l.: s.n.], 2004. p. 210–215.

ZHONG, H.; WONHAM, W. M. On the consistency of hierarchical supervision in discrete-event systems. **IEEE Trans. on Automatic Control**, v. 35, n. 10, p. 1125–1134, Oct 1990. ISSN 0018-9286.

ZOITL, A. et al. Executing real-time constrained control applications modelled in iec 61499 with respect to dynamic reconfiguration. In: **Industrial Informatics, 2005. INDIN '05. 2005 3rd IEEE International Conference on**. [S.l.: s.n.], 2005. p. 62 – 67.

ZOITL, A.; SUNDER, C.; TERZIC, I. Dynamic reconfiguration of distributed control applications with reconfiguration services based on IEC 61499. In: **Distributed Intelligent Systems: Collective Intelligence and Its Applications, 2006. DIS 2006. IEEE Workshop on**. Prague: IEEE, 2006. p. 109–114.

APÊNDICE A – BLOCO DE FUNÇÃO DA LINHA DE PRODUÇÃO

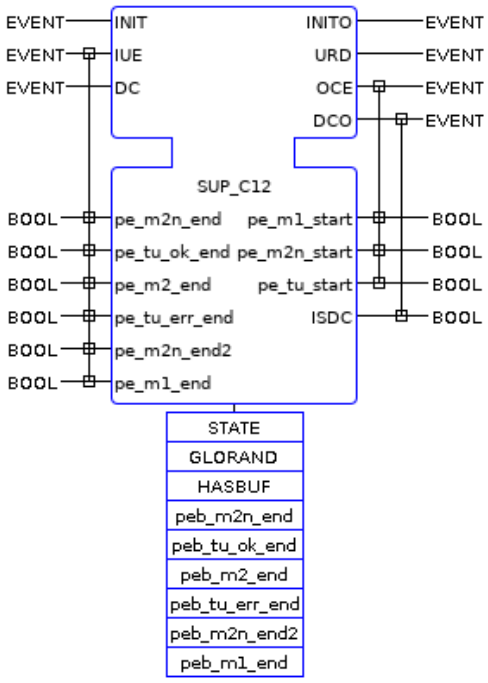
Neste apêndice é mostrado o FB da linha de produção da Figura 35b (Página 77), gerado automaticamente a partir do modelo na Figura 38 (Página 79). O método de geração é o apresentado na Seção 6.3. Assim, tem-se na Figura 48 a interface deste FB. O Algoritmo A.1 mostra o *xml* deste FB.

Como os *labels* do modelo (Figura 38) são pares e a variável que armazena o estado no FB é um inteiro, então o *label* dos estados são convertidos da seguinte maneira, sendo p o primeiro elemento do par e s o segundo elemento do par:

$$label = \begin{cases} p & \text{se } p \neq u \\ -s & \text{se } p = u \end{cases}$$

Por exemplo, um estado com o *label* $(2, 3)$ passará a ter o *label* 2. Um estado com o *label* $(u, 4)$ passará a ter o *label* -4.

Figura 48 – Uma representação genérica de uma FBN para implementar o modelo de um controlador.



Fonte: Próprio Autor

Algoritmo A.1 – O xml do FB mostrado na Figura 48.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<FBType Name="SUP_C12" Comment="Basic Function Block Type" >
  <Identification Standard="61499-2" />
```

```
  <VersionInfo Organization="UDESC" Version="0.3" Author="Generated by Rutt Reconf Tool"
  Date="2018-05-05" />
<CompilerInfo header="package fb.rt.safereconf;" >
</CompilerInfo>
  <InterfaceList>
    <EventInputs>
      <Event Name="INIT" Comment="Initialization Request" >
      </Event>
      <Event Name="IUE" Comment="Test the uncontrollable input" >
        <With Var="pe_m2n_end" />
        <With Var="pe_tu_ok_end" />
        <With Var="pe_m2_end" />
        <With Var="pe_tu_err_end" />
        <With Var="pe_m2n_end2" />
        <With Var="pe_m1_end" />
      </Event>
      <Event Name="DC" Comment="Set/reset a flag to disable some controllables defined
      in the automaton" >
      </Event>
    </EventInputs>
    <EventOutputs>
      <Event Name="INITO" Comment="Initialization Confirm" >
      </Event>
      <Event Name="URD" Comment="Read uncontrollable events again" >
      </Event>
      <Event Name="OCE" Comment="Controllable event generated" >
        <With Var="pe_m1_start" />
        <With Var="pe_m2n_start" />
        <With Var="pe_tu_start" />
      </Event>
      <Event Name="DCO" >
        <With Var="ISDC" />
      </Event>
    </EventOutputs>
    <InputVars>
      <VarDeclaration Name="pe_m2n_end" Type="BOOL" Comment="Input event qualifier" />
      <VarDeclaration Name="pe_tu_ok_end" Type="BOOL" Comment="Input event qualifier" />
      <VarDeclaration Name="pe_m2_end" Type="BOOL" Comment="Input event qualifier" />
      <VarDeclaration Name="pe_tu_err_end" Type="BOOL" Comment="Input event qualifier"/>
      <VarDeclaration Name="pe_m2n_end2" Type="BOOL" Comment="Input event qualifier" />
      <VarDeclaration Name="pe_m1_end" Type="BOOL" Comment="Input event qualifier" />
    </InputVars>
    <OutputVars>
      <VarDeclaration Name="pe_m1_start" Type="BOOL" Comment="Input event qualifier" />
      <VarDeclaration Name="pe_m2n_start" Type="BOOL" Comment="Input event qualifier" />
      <VarDeclaration Name="pe_tu_start" Type="BOOL" Comment="Input event qualifier" />
      <VarDeclaration Name="ISDC" Type="BOOL" Comment="If true then some controllables
      are disabled" />
    </OutputVars>
  </InterfaceList>
  <BasicFB>
    <InternalVars>
      <VarDeclaration Name="STATE" Type="SINT" />
      <VarDeclaration Name="GLORAND" Type="INT" />
      <VarDeclaration Name="HASBUF" Type="INT" />
      <VarDeclaration Name="peb_m2n_end" Type="INT" Comment="Input event qualifier" />
      <VarDeclaration Name="peb_tu_ok_end" Type="INT" Comment="Input event qualifier" />
      <VarDeclaration Name="peb_m2_end" Type="INT" Comment="Input event qualifier" />
      <VarDeclaration Name="peb_tu_err_end" Type="INT" Comment="Input event qualifier" />
```

```
      <VarDeclaration Name="peb_m2n_end2" Type="INT" Comment="Input event qualifier" />
      <VarDeclaration Name="peb_m1_end" Type="INT" Comment="Input event qualifier" />
    </InternalVars>
    <ECC >
      <ECState Name="START" Comment="Initial State" x="2061.111" y="350.0" >
      </ECState>
      <ECState Name="ST_INIT" x="1544.4443" y="105.55555" >
        <ECAction Algorithm="INIT" Output="INITO" />
      </ECState>
      <ECState Name="ST_HANDLE_UC" x="2016.6666" y="600.0" >
        <ECAction Algorithm="ALG_HANDLE_UC" />
      </ECState>
      <ECState Name="ST_GEN_C" x="1561.1111" y="899.99994" >
        <ECAction Algorithm="ALG_GEN_C" Output="OCE" />
      </ECState>
      <ECState Name="ST_UCREAD" x="2577.7776" y="899.99994" >
        <ECAction Output="URD" />
      </ECState>
      <ECState Name="STDC" x="2466.6665" y="261.1111" >
        <ECAction Algorithm="ALG_DC" Output="DCO" />
      </ECState>
      <ECTransition Source="START" Destination="ST_INIT" Condition="INIT" x="1794.4443"
      y="261.1111" />
      <ECTransition Source="ST_INIT" Destination="START" Condition="1" x="1544.4443"
      y="350.0" />
      <ECTransition Source="START" Destination="ST_HANDLE_UC" Condition="IUE"
      x="2016.6666" y="455.55554" />
      <ECTransition Source="ST_HANDLE_UC" Destination="ST_GEN_C" Condition="
      [HASBUF = 0]"
      x="1777.7777" y="727.7778" />
      <ECTransition Source="ST_GEN_C" Destination="START" Condition="1" x="1561.1111"
      y="522.2222" />
      <ECTransition Source="ST_HANDLE_UC" Destination="ST_UCREAD" Condition="
      [HASBUF > 0]" x="2577.7776" y="716.6666" />
      <ECTransition Source="ST_UCREAD" Destination="START" Condition="1" x="3188.8887"
      y="605.55554" />
      <ECTransition Source="START" Destination="STDC" Condition="DC" x="2166.6665"
      y="172.22221" />
      <ECTransition Source="STDC" Destination="START" Condition="1" x="2288.889"
      y="344.44443" />
    </ECC>
  <Algorithm Name="INIT" >
<Other Language="Java"><![CDATA[STATE.value = 0;
HASBUF.value = 0;
ISDC.value = false;
pe_m2n_end.value = false;
peb_m2n_end.value = 0;
pe_m1_start.value = false;
pe_tu_ok_end.value = false;
peb_tu_ok_end.value = 0;
pe_m2_end.value = false;
peb_m2_end.value = 0;
pe_tu_err_end.value = false;
peb_tu_err_end.value = 0;
pe_m2n_start.value = false;
pe_tu_start.value = false;
pe_m2n_end2.value = false;
peb_m2n_end2.value = 0;
pe_m1_end.value = false;
```

```
peb_m1_end.value = 0;
]]></Other>
  </Algorithm>
  <Algorithm Name="ALG_HANDLE_UC" >
<Other Language="Java"><![CDATA[if(pe_m2n_end.value){ peb_m2n_end.value++;
pe_m2n_end.value = false; HASBUF.value++;}
if(pe_tu_ok_end.value){ peb_tu_ok_end.value++; pe_tu_ok_end.value = false;
HASBUF.value++;}
if(pe_m2_end.value){ peb_m2_end.value++; pe_m2_end.value = false; HASBUF.value++;}
if(pe_tu_err_end.value){ peb_tu_err_end.value++; pe_tu_err_end.value = false;
HASBUF.value++;}
if(pe_m2n_end2.value){ peb_m2n_end2.value++; pe_m2n_end2.value = false; HASBUF.value++;}
if(pe_m1_end.value){ peb_m1_end.value++; pe_m1_end.value = false; HASBUF.value++;}


if(HASBUF.value > 0){
if(STATE.value == 24){
 if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=8;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=8;}
else    if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=26;}
}else if(STATE.value == 26){
 if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=11;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=11;}
}else if(STATE.value == 20){
 if(peb_m2_end.value > 0){peb_m2_end.value--; STATE.value=24;}
else    if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=23;}
else    if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=5;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=5;}
}else if(STATE.value == 21){
 if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=6;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=6;}
}else if(STATE.value == 23){
 if(peb_m2_end.value > 0){peb_m2_end.value--; STATE.value=26;}
else    if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=7;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=7;}
}else if(STATE.value == -20){
 if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=-23;}
else    if(peb_m2n_end2.value > 0){peb_m2n_end2.value--; STATE.value=12;}
else    if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=-5;}
else    if(peb_m2n_end.value > 0){peb_m2n_end.value--; STATE.value=24;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=-5;}
}else if(STATE.value == -23){
 if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=-7;}
else    if(peb_m2n_end2.value > 0){peb_m2n_end2.value--; STATE.value=15;}
else    if(peb_m2n_end.value > 0){peb_m2n_end.value--; STATE.value=26;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=-7;}
}else if(STATE.value == 1){
 if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=2;}
}else if(STATE.value == 0){
}else if(STATE.value == 2){
}else if(STATE.value == 5){
 if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=7;}
else    if(peb_m2_end.value > 0){peb_m2_end.value--; STATE.value=8;}
}else if(STATE.value == 4){
 if(peb_m2_end.value > 0){peb_m2_end.value--; STATE.value=6;}
}else if(STATE.value == 7){
 if(peb_m2_end.value > 0){peb_m2_end.value--; STATE.value=11;}
}else if(STATE.value == 6){
}else if(STATE.value == 9){
```

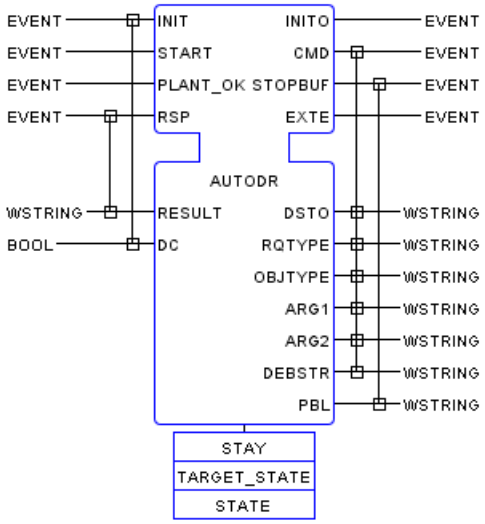
```
 if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=0;}
else    if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=0;}
}else if(STATE.value == 8){
 if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=11;}
}else if(STATE.value == -18){
 if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=-4;}
else    if(peb_m2n_end2.value > 0){peb_m2n_end2.value--; STATE.value=9;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=-4;}
else    if(peb_m2n_end.value > 0){peb_m2n_end.value--; STATE.value=21;}
}else if(STATE.value == 11){
}else if(STATE.value == 12){
 if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=15;}
else    if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=1;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=1;}
}else if(STATE.value == 15){
 if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=2;}
else    if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=2;}
}else if(STATE.value == 18){
 if(peb_m2_end.value > 0){peb_m2_end.value--; STATE.value=21;}
else    if(peb_tu_err_end.value > 0){peb_tu_err_end.value--; STATE.value=4;}
else    if(peb_tu_ok_end.value > 0){peb_tu_ok_end.value--; STATE.value=4;}
}else if(STATE.value == -5){
 if(peb_m1_end.value > 0){peb_m1_end.value--; STATE.value=-7;}
else    if(peb_m2n_end2.value > 0){peb_m2n_end2.value--; STATE.value=1;}
else    if(peb_m2n_end.value > 0){peb_m2n_end.value--; STATE.value=8;}
}else if(STATE.value == -4){
 if(peb_m2n_end2.value > 0){peb_m2n_end2.value--; STATE.value=0;}
else    if(peb_m2n_end.value > 0){peb_m2n_end.value--; STATE.value=6;}
}else if(STATE.value == -7){
 if(peb_m2n_end2.value > 0){peb_m2n_end2.value--; STATE.value=2;}
else    if(peb_m2n_end.value > 0){peb_m2n_end.value--; STATE.value=11;}
}
HASBUF.value--;
}

]]></Other>
  </Algorithm>
  <Algorithm Name="ALG_GEN_C" >
<Other Language="Java"><![CDATA[pe_m1_start.value = false;
pe_m2n_start.value = false;
pe_tu_start.value = false;
if(STATE.value == 24){
}
else if(STATE.value == 26){
}
else if(STATE.value == 20){
}
else if(STATE.value == 21){
pe_m1_start.value = true; STATE.value = 24;
}
else if(STATE.value == 23){
}
else if(STATE.value == -20){
}
else if(STATE.value == -23){
}
else if(STATE.value == 1){
}
else if(STATE.value == 0){
```

```
pe_m1_start.value = true; STATE.value = 1;
}
else if(STATE.value == 2){
if(!ISDC.value){ pe_m2n_start.value = true; STATE.value = -4;}
}
else if(STATE.value == 5){
}
else if(STATE.value == 4){
pe_m1_start.value = true; STATE.value = 5;
}
else if(STATE.value == 7){
}
else if(STATE.value == 6){
int gen = GLORAND.value % 2;
if (gen == 0){  pe_m1_start.value = true; STATE.value = 8;}
if (gen == 1){  pe_tu_start.value = true; STATE.value = 9;}
}
else if(STATE.value == 9){
pe_m1_start.value = true; STATE.value = 12;
}
else if(STATE.value == 8){
pe_tu_start.value = true; STATE.value = 12;
}
else if(STATE.value == -18){
pe_m1_start.value = true; STATE.value = -20;
}
else if(STATE.value == 11){
pe_tu_start.value = true; STATE.value = 15;
}
else if(STATE.value == 12){
}
else if(STATE.value == 15){
if(!ISDC.value){ pe_m2n_start.value = true; STATE.value = -18;}
}
else if(STATE.value == 18){
pe_m1_start.value = true; STATE.value = 20;
}
else if(STATE.value == -5){
}
else if(STATE.value == -4){
pe_m1_start.value = true; STATE.value = -5;
}
else if(STATE.value == -7){
}
GLORAND.value++; if(GLORAND.value > 100000) GLORAND.value = 0;
]]></Other>
  </Algorithm>
  <Algorithm Name="ALG_DC" >
<ST><![CDATA[ISDC := NOT ISDC;]]></ST>
  </Algorithm>
  </BasicFB>
</FBType>
```

APÊNDICE B – BLOCO DE FUNÇÃO AUTODR

Neste apêndice é mostrado o FB da aplicação de reconfiguração dinâmica gerado automaticamente. O método de geração é o apresentado na Seção 6.3. Assim, tem-se na Figura 49 a interface deste FB. O Algoritmo B.1 mostra o *xml* deste FB.

Figura 49 – Interface do FB AUTODR.



Fonte: Próprio Autor

Algoritmo B.1 – O xml do FB mostrado na Figura 49.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
<FBType Name="AUTODR" Comment="Basic Function Block Type" >
  <Identification Standard="61499-2" />
  <VersionInfo Organization="Holobloc Inc" Version="0.3" Author="JHC" Date="2009-09-15"
    Remarks="Changed transition priorities." />
  <VersionInfo Organization="Rockwell Automation" Version="0.2" Author="JHC" Date="2003-12-04"
    Remarks="Renamed for better indexing." />
  <VersionInfo Organization="Rockwell Automation" Version="0.1" Author="JHC" Date="2002-11-05"
    Remarks="Corrected missing ST element in Algorithm REQ." />
  <VersionInfo Organization="Rockwell Automation" Version="0.0" Author="JHC"
    Date="2000-05-30" />
<CompilerInfo header="package fb.rt.safereconf;" >
</CompilerInfo>
  <InterfaceList>
    <EventInputs>
      <Event Name="INIT" Comment="Initialization Request" >
        <With Var="DC" />
      </Event>
      <Event Name="START" Comment="Normal Execution Request" >
      </Event>
      <Event Name="PLANT_OK" >
      </Event>
```

```
      <Event Name="RSP" >
        <With Var="RESULT" />
      </Event>
    </EventInputs>
    <EventOutputs>
      <Event Name="INITO" Comment="Initialization Confirm" >
      </Event>
      <Event Name="CMD" Comment="Execution Confirmation" >
        <With Var="DSTO" />
        <With Var="RQTYPE" />
        <With Var="OBJTYPE" />
        <With Var="ARG1" />
        <With Var="ARG2" />
        <With Var="DEBSTR" />
      </Event>
      <Event Name="STOPBUF" >
        <With Var="PBL" />
      </Event>
      <Event Name="EXTE" >
      </Event>
    </EventOutputs>
    <InputVars>
      <VarDeclaration Name="RESULT" Type="WSTRING" Comment="Input event qualifier" />
      <VarDeclaration Name="DC" Type="BOOL" />
    </InputVars>
    <OutputVars>
      <VarDeclaration Name="DSTO" Type="WSTRING" Comment="Output event qualifier" />
      <VarDeclaration Name="RQTYPE" Type="WSTRING" />
      <VarDeclaration Name="OBJTYPE" Type="WSTRING" />
      <VarDeclaration Name="ARG1" Type="WSTRING" />
      <VarDeclaration Name="ARG2" Type="WSTRING" />
      <VarDeclaration Name="DEBSTR" Type="WSTRING" />
      <VarDeclaration Name="PBL" Type="WSTRING" />
    </OutputVars>
  </InterfaceList>
  <BasicFB>
    <InternalVars>
      <VarDeclaration Name="STAY" Type="BOOL" />
      <VarDeclaration Name="TARGET_STATE" Type="INT" />
      <VarDeclaration Name="STATE" Type="INT" />
    </InternalVars>
    <ECC >
      <ECState Name="START" Comment="Initial State" x="2211.111" y="1111.1111" >
      </ECState>
      <ECState Name="INIT" Comment="Initialization" x="2522.2222" y="888.88885" >
        <ECAction Algorithm="INIT" Output="INITO" />
      </ECState>
      <ECState Name="STOP" x="1844.4443" y="1288.8888" >
        <ECAction Output="STOPBUF" />
      </ECState>
      <ECState Name="REPLACE" x="1583.3333" y="1488.8888" >
        <ECAction Algorithm="ALG_REPLACE" Output="CMD" />
      </ECState>
      <ECState Name="ENABLE_DC" x="2227.7776" y="3050.0" >
        <ECAction Algorithm="ALG_E_DC" Output="CMD" />
      </ECState>
      <ECState Name="STARTBUF" x="1788.8888" y="2566.6665" >
        <ECAction Output="STOPBUF" />
        <ECAction Output="EXTE" />
```

```
      </ECState>
      <ECState Name="END" x="2683.3333" y="2622.2222" >
        <ECAction Output="EXTE" />
      </ECState>
      <ECState Name="RST" x="1572.2222" y="1849.9999" >
        <ECAction Algorithm="ALG_RST" />
      </ECState>
      <ECState Name="SUPINIT" x="2155.5554" y="2027.7777" >
        <ECAction Output="EXTE" />
      </ECState>
      <ECState Name="SUPSET" x="1788.8888" y="2194.4443" >
        <ECAction Algorithm="ALG_SETINT" Output="CMD" />
      </ECState>
      <ECState Name="RDEVT" x="1177.7777" y="2883.3333" >
        <ECAction Algorithm="ALG_READEVT" Output="CMD" />
      </ECState>
      <ECTransition Source="INIT" Destination="START" Condition="1" x="2522.2222"
        y="1111.1111" />
      <ECTransition Source="START" Destination="INIT" Condition="INIT" x="2211.111"
        y="888.88885" />
      <ECTransition Source="START" Destination="STOP" Condition="START" x="1844.4443"
        y="1111.1111" />
      <ECTransition Source="REPLACE" Destination="REPLACE" Condition="RSP [STAY]" x="1350.0"
        y="1327.7777" />
      <ECTransition Source="ENABLE_DC" Destination="END" Condition="RSP [NOT STAY]"
        x="2683.3333" y="2855.5554" />
      <ECTransition Source="ENABLE_DC" Destination="ENABLE_DC" Condition="RSP [STAY]"
        x="2227.7776" y="3238.8887" />
      <ECTransition Source="END" Destination="START" Condition="1"
        x="2683.3333" y="1277.7777" />
      <ECTransition Source="STOP" Destination="REPLACE" Condition="1"
        x="1583.3333" y="1288.8888" />
      <ECTransition Source="STARTBUF" Destination="ENABLE_DC" Condition="PLANT_OK"
        x="2227.7776" y="2777.7776" />
      <ECTransition Source="REPLACE" Destination="RST" Condition="START"
        x="1277.7777" y="1644.4443" />
      <ECTransition Source="RST" Destination="REPLACE" Condition="1"
        x="1583.3333" y="1705.5555" />
      <ECTransition Source="REPLACE" Destination="SUPINIT" Condition="RSP [NOT STAY]"
        x="2155.5554" y="1638.8888" />
      <ECTransition Source="SUPSET" Destination="STARTBUF" Condition="RSP [NOT STAY]"
        x="1788.8888" y="2350.0" />
      <ECTransition Source="SUPSET" Destination="SUPSET" Condition="RSP [STAY]"
        x="1394.4443" y="2194.4443" />
      <ECTransition Source="SUPINIT" Destination="SUPSET" Condition="1"
        x="1788.8888" y="2027.7777" />
      <ECTransition Source="STARTBUF" Destination="RDEVT" Condition="START"
        x="1177.7777" y="2566.6665" />
      <ECTransition Source="RDEVT" Destination="RDEVT" Condition="START"
        x="1088.8888" y="3050.0" />
      <ECTransition Source="RDEVT" Destination="ENABLE_DC" Condition="PLANT_OK"
        x="1472.2222" y="3050.0" />
    </ECC>
  <Algorithm Name="INIT" Comment="Initialization algorithm" >
<ST><![CDATA[DSTO:="";
RQTYPE:="";
OBJTYPE:="";
ARG1:="";
ARG2:="";
```

```
DEBSTR:="";
PBL:="";
RESULT:="";
DC:=FALSE;
STATE:=0;]]></ST>
  </Algorithm>
  <Algorithm Name="ALG_REPLACE" >
<Other Language="Java"><![CDATA[
DSTO.value = "PN1";
STAY.value = true;
if (STATE.value == 0) {
 RQTYPE.value = "STOP";
 OBJTYPE.value = "FB";
 ARG1.value = "sup_c1";
 ARG2.value = "SUP_C1";
 STATE.value = 1;
 DEBSTR.value = "Stopping C1\n";
} else if (STATE.value == 1) {
 RQTYPE.value = "READ";
 OBJTYPE.value = "Parameter";
 ARG1.value = "sup_c1.STATE";
 ARG2.value = "";
 STATE.value = 2;
 DEBSTR.value = "Reading current state...";
} else if (STATE.value == 2) {
 int p = RESULT.value.indexOf("Value");
 p = p + 7;
 int l = RESULT.value.lastIndexOf("\"");
 String w = RESULT.value.substring(p, l);
 TARGET_STATE.value = Short.parseShort(w);
 DEBSTR.value = DEBSTR.value + "is " + w + "\n";
 RQTYPE.value = "DELETE";
 OBJTYPE.value = "FB";
 ARG1.value = "sup_c1";
 ARG2.value = "SUP_C1";
 STATE.value = 3;
 DEBSTR.value = DEBSTR.value + "Excluindo SUP_C1\n";
} else if (STATE.value == 3) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "FB";
 ARG1.value = "sup_c12";
 ARG2.value = "SUP_C12";
 STATE.value = 4;
 DEBSTR.value = "Criando SUP_C12\n";
} else if (STATE.value == 4) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "m1_end.QO";
 ARG2.value = "sup_c12.pe_m1_end";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 5) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "m2_end.QO";
 ARG2.value = "sup_c12.pe_m2_end";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 6) {
```

```
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "tu_ok_end.QO";
 ARG2.value = "sup_c12.pe_tu_ok_end";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 7) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "tu_err_end.QO";
 ARG2.value = "sup_c12.pe_tu_err_end";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 8) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "m2n_end.QO";
 ARG2.value = "sup_c12.pe_m2n_end";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 9) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "m2n_end2.QO";
 ARG2.value = "sup_c12.pe_m2n_end2";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 10) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "breakthecicle1.EO";
 ARG2.value = "sup_c12.IUE";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 11) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "break2.EO";
 ARG2.value = "sup_c12.IUE";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 12) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "sup_c12.URD";
 ARG2.value = "break2.START";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 13) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "sup_c12.OCE";
 ARG2.value = "sup_merge.EO";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 14) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "sup_c12.pe_m1_start";
```

```
 ARG2.value = "m1_start.PERMIT";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 15) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "sup_c12.pe_m2n_start";
 ARG2.value = "m2n_start.PERMIT";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 16) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "sup_c12.pe_tu_start";
 ARG2.value = "tu_start.PERMIT";
 STATE.value++;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 17) {
 RQTYPE.value = "START";
 OBJTYPE.value = "FB";
 ARG1.value = "sup_c12";
 ARG2.value = "SUP_C12";
 STATE.value++;
 DEBSTR.value = "Starting sup_c12\n";
} else if (STATE.value == 18) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "SUB_EXTEVT.IND";
 ARG2.value = "sup_c12.INIT";
 STATE.value = 0;
 DEBSTR.value = "Criando " + ARG1.value + " -> " + ARG2.value + "\n";
 STAY.value = false;
}]]></Other>
  </Algorithm>
  <Algorithm Name="ALG_E_DC" >
<Other Language="Java"><![CDATA[DSTO.value = "PN1";
STAY.value = true;
RQTYPE.value = "WRITE";
OBJTYPE.value = "Parameter";
ARG1.value = "sup_c12.ISDC";
ARG2.value = "FALSE";
STATE.value=0;
DEBSTR.value = "Enabling all events\n";
DC.value=true;
STAY.value = false;]]></Other>
  </Algorithm>
  <Algorithm Name="ALG_RST" >
<Other Language="Java"><![CDATA[STATE.value=0;]]></Other>
  </Algorithm>
  <Algorithm Name="ALG_SETINT" >
<Other Language="Java"><![CDATA[
DSTO.value = "PN1";
STAY.value = true;
if (STATE.value == 0) {
 RQTYPE.value = "WRITE";
 OBJTYPE.value = "Parameter";
 ARG1.value = "sup_c12.STATE";
 ARG2.value = String.valueOf(TARGET_STATE.value);
 STATE.value++;
```

```
 DEBSTR.value = "Set sup_c12 State to " + String.valueOf(TARGET_STATE.value) + "\n";
} else if (STATE.value == 1) {
 RQTYPE.value = "WRITE";
 OBJTYPE.value = "Parameter";
 ARG1.value = "sup_c12.ISDC";
 ARG2.value = "TRUE";
 STATE.value++;
 DEBSTR.value = "Disabling some controllable events.\n";
 DC.value = true;
} else if (STATE.value == 2) {
 RQTYPE.value = "DELETE";
 OBJTYPE.value = "Connection";
 ARG1.value = "SUB_EXTEVT.IND";
 ARG2.value = "sup_c12.INIT";
 STATE.value++;
 DEBSTR.value = "Deleting " + ARG1.value + " -> " + ARG2.value + "\n";
} else if (STATE.value == 3) {
 RQTYPE.value = "CREATE";
 OBJTYPE.value = "Connection";
 ARG1.value = "SUB_EXTEVT.IND";
 ARG2.value = "breakthecicle1.START";
 STATE.value = 0;
 DEBSTR.value = "Creating " + ARG1.value + " -> " + ARG2.value + "\n";
 STAY.value = false;
}]]></Other>
  </Algorithm>
  <Algorithm Name="ALG_READEVT" >
<Other Language="Java"><![CDATA[RQTYPE.value = "READ";
OBJTYPE.value = "Parameter";
ARG1.value = "sup_c12.STATE";
ARG2.value = "";
DEBSTR.value = "Reading current state...";]]></Other>
  </Algorithm>
  </BasicFB>
</FBType>
```