Este trabalho apresenta uma arquitetura de diagnóstico multicamadas para detecção de falhas em sistemas embarcados, que permite uma melhor discriminação do tipo e origem da falha, possibilitando um diagnóstico mais preciso e assertivo. Os diagnosticadores são projetados conforme a metodologia de diagnóstico de falhas em SEDs modelados por autômatos. Uma vez concebidos, os diagnosticadores são implementados em linguagem ANSI C, através de uma ferramenta de geração automática de software, e finalmente incorporados ao software principal do equipamento onde se pretende realizar o diagnóstico. Esta arquitetura de diagnóstico foi aplicada em um estudo de caso para um refrigerador *Frost Free*, para o qual foram projetados os diagnosticadores, em seguida os mesmos foram implementados em software e posteriormente validados a fim de comprovar a eficácia não somente dos diagnosticadores mas também da arquitetura proposta, além do processo de conversão dos mesmos em linguagem de software.

Orientador: Prof. Dr. André Bittencourt Leal

Joinville, 2014

ANO 2014

DANIEL GUMIERO NORONHA MAAS | DIAGNÓSTICO DE FALHAS MULTICAMADAS DE SISTEMAS EMBARCADOS MODELADOS POR SEDS

UDESC

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
CURSO DE MESTRADO PROFISSIONAL EM ENGENHARIA ELÉTRICA

DISSERTAÇÃO DE MESTRADO

DIAGNÓSTICO DE FALHAS MULTICAMADAS DE SISTEMAS EMBARCADOS MODELADOS POR SEDS

DANIEL GUMIERO NORONHA MAAS

JOINVILLE, 2014

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM ENGENHARIA ELÉTRICA

DANIEL GUMIERO NORONHA MAAS

DIAGNÓSTICO DE FALHAS MULTICAMADAS DE SISTEMAS EMBARCADOS MODELADOS POR SEDS

JOINVILLE

2014

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM ENGENHARIA ELÉTRICA

DANIEL GUMIERO NORONHA MAAS

DIAGNÓSTICO DE FALHAS MULTICAMADAS DE SISTEMAS EMBARCADOS MODELADOS POR SEDS

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, para a obtenção do grau de Mestre em Engenharia Elétrica.

Orientador:
Prof. Dr. André Bittencourt Leal

JOINVILLE

2014

FICHA CATALOGRÁFICA

M111d Maas, Daniel Gumiero Noronha
Diagnóstico de falhas multicamadas de sistemas embarcados modelados por SEDs/ Daniel Gumiero Noronha Maas. – 2014.
173 p. : il. ; 21 cm

Orientador: André Bittencourt Leal
Bibliografia: p. 173-177
Dissertação (mestrado) – Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Mestrado em Engenharia Elétrica, Joinville, 2014.

1. Engenharia elétrica. 2. Diagnóstico de falhas. 3. Sistemas embarcados. 4. Sistemas a eventos discretos. I. Leal, André Bittencourt. II. Universidade do Estado de Santa Catarina. Programa de Pós-graduação em Engenharia Elétrica. III. Título.

CDD: 621.3 – 20.ed.

DANIEL GUMIERO NORONHA MAAS

DIAGNÓSTICO DE FALHAS MULTICAMADAS DE SISTEMAS EMBARCADOS MODELADOS POR SEDS

Dissertação apresentada ao Curso de Mestrado Profissional em Engenharia Elétrica como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica na área de concentração "Automação de Sistemas".

Banca Examinadora

Orientador:

Prof. Dr. André Bittencourt Leal
CCT/UDESC

Membros

Prof. Dr. João Carlos dos Santos Basilio
COPPE/UFRJ

Prof. Dr. Ademir Nied
CCT/UDESC

Joinville, 09 de setembro de 2014.

Este trabalho é dedicado aos meus filhos João Victor, Pedro Henrique e Maria Victória, e especialmente à minha esposa Érika, pelo apoio, compreensão e imensa paciência.

AGRADECIMENTOS

Gostaria de agradecer sobre tudo a *Deus* por ter me dado forças e saúde para vencer mais este desafio.

Agradeço grandemente à minha esposa *Érika Maas Noronha*, por sempre acreditar em mim.

Muito obrigado aos meus pais *Elisabete Gumiero Noronha* e *José Carlos Noronha*, que mesmo distantes, sempre me apoiaram e me deram forças para alcançar esse objetivo.

Muito obrigado aos meus sogros *Rolfe Maas* e *Maria das Graças e Silva Maas*, por sempre acreditarem em mim, pelas palavras de incentivo e suporte nos momentos difíceis.

Muito obrigado ao grande amigo *Cláudio Navarro*, sempre presente e disposto a ajudar e compartilhar seus conhecimentos e experiências.

Muito obrigado ao meu orientador Prof. Dr. *André Bittencourt Leal*, por seu comprometimento, apoio e também por ter abdicado de momentos com sua família para se dedicar à revisão deste trabalho.

Muito obrigado aos professores membros da banca examinadora, professor Dr. *João Carlos dos Santos Basilio* e professor Dr. *Ademir Nied* por seu tempo dedicado e suas enriquecedoras contribuições.

Muito obrigado aos colegas de trabalho *Valmor Adami Junior*, *Alexandre Junkes Pinotti*, *Carlos Henrique Nacer* e *Luiz Felipe da Silva* por todas ideias e discussões que geraram diversas contribuições. Em nome desses meu obrigado à *Whirlpool* por permitir que desenvolvesse meu trabalho.

A todos meu Muito Obrigado.

Jamais considere seus estudos como uma obrigação, mas como uma oportunidade invejável para aprender a conhecer a influência libertadora da beleza do reino do espírito, para seu próprio prazer pessoal e para proveito da comunidade à qual seu futuro trabalho pertencer.
(Albert Einstein)

RESUMO

Este trabalho apresenta uma arquitetura de diagnóstico multicamadas para detecção de falhas em sistemas embarcados, que permite uma melhor discriminação do tipo e origem da falha, possibilitando um diagnóstico mais preciso e assertivo. Esta arquitetura contempla as interfaces necessárias para permitir a integração no sistema embarcado e também considera o tratamento das informações de diagnóstico para fins de ações de recuperação do sistema, ou simplesmente a externalização destas informações. Neste trabalho, consideram-se os diagnosticadores projetados conforme a metodologia de diagnóstico de falhas em SEDs modelados por autômatos. Uma vez concebidos, os diagnosticadores são implementados em linguagem ANSI C, através de uma ferramenta de geração automática de software, e finalmente incorporados ao software principal do equipamento onde se pretende realizar o diagnóstico. Esta arquitetura de diagnóstico foi então aplicada em um estudo de caso para um refrigerador *Frost Free*, para o qual foram projetados os diagnosticadores, em seguida os mesmos foram implementados em software e posteriormente validados a fim de comprovar a eficácia não somente dos diagnosticadores mas também da arquitetura proposta, além do processo de conversão dos mesmos em linguagem de software.

Palavras-chaves: diagnóstico de falhas, sistemas embarcados, sistemas a eventos discretos.

ABSTRACT

This work presents a multilayer architecture for fault diagnosis in embedded systems that allows a better discrimination of the type and source of the failure, providing an accurate and assertive diagnosis. This architecture contemplates the necessary interfaces to allow integration of this diagnostic framework in the embedded system and also considers the treatment of diagnostic information for recovery system actions purposes, or simply allows the externalization of such information. This work considers the diagnosers designed according to the methodology of fault diagnosis in DES modeled by automata. Once designed the diagnosers are implemented in ANSI C language through an automated software generation tool, and finally incorporated into the main product software where it intends to perform the diagnosis. This architecture diagnosis was then applied in a case study for Frost Free refrigerator, for which the diagnosers were designed then were implemented in software and subsequently validated in order to confirm the effectiveness of the diagnosers, of the proposed architecture beyond the C language conversion process.

Key-words: fault diagnosis, embedded systems, discrete event systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ANSI	American National Standards Institute
API	Application Programming Interface
DAM	Diagnoser Automata Model
GF	Gerenciador de Falhas
IDE	Integrated Development Environment
IDES	Integrated Discrete-Event Systems
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
MIT	Massachusetts Institute of Technology
NTC	Negative Temperature Coefficiente
RC	Refrigerator Compartment
SED	Sistema a Eventos Discretos
SRF	Sistema de Recuperação de Falhas
STVD	ST Visual Develop

LISTA DE SÍMBOLOS

Σ	Conjunto de eventos
Σ_o	Conjunto de eventos observáveis, $\Sigma_o \subset \Sigma$
Σ_{uo}	Conjunto de eventos não-observáveis, $\Sigma_{uo} \subset \Sigma$
$\Sigma_f = \{ \sigma_f \}$	Conjunto de eventos de falha, $\Sigma_f \subseteq \Sigma_{uo}$
Σ^*	Fecho de Kleene
L	Linguagem gerada por um autômato
$\overline{s}$	Prefixo Fechamento de uma sequência s
$\| t \|$	Comprimento de uma sequência t
L/s	Continuação da linguagem L após uma sequência s
$\Psi(\Sigma_f)$	Conjunto de todas as sequências de L que terminam com um evento de Σ_f
$\Sigma_f \in s$	$\overline{s} \cap \Psi(\Sigma_f) \neq \emptyset$
2^A	Conjunto potência de A
$\setminus$	Operador de remoção de eventos de um conjunto
P_o	Projeção de elementos de Σ^* em Σ_o^*
P_o^{-1}	Projeção inversa de elementos de Σ_o^* em 2^{Σ^*}
$G_1 \parallel G_2$	Composição paralela de G_1 e G_2
$Obs(G,\Sigma_o)$	Observador de G em relação a Σ_o
G_d	Diagnosticador centralizado para o autômato G
A_l	Autômato rotulador de falhas

SUMÁRIO

1 INTRODUÇÃO

O constante aumento da complexidade dos sistemas tecnológicos e a crescente exigência por desempenho e confiabilidade, tem demandado o desenvolvimento de métodos cada vez mais sofisticados e sistemáticos para um diagnóstico rápido e preciso das falhas que podem ocorrer no sistema (SAMPATH et al., 1996; CARVALHO, 2011). Perante esta realidade, o problema de diagnóstico de falhas, tem recebido atenção considerável em diversas áreas dentre as quais podemos citar a engenharia de confiabilidade, controle e ciência da computação, e uma variedade de propostas tem surgido, dentre as quais destacam-se os métodos utilizando árvores de falhas e as metodologias que usam bases de conhecimento extraídas dos operadores do processo e dos dados obtidos do processo real (ISERMAN, 1997). Nessas metodologias, a detecção de falhas podem ser feitas por geração de sintomas analíticos, heurísticos e diagnóstico de falhas. Na geração de sintomas analíticos, os dados coletados do processo são usados para produzir informação quantificável e analisável. Os sintomas heurísticos, que normalmente são representados por variáveis linguística tais como: pequeno, médio, grande, quente, frio, etc., podem ser obtidos dos operadores da planta ou processo, por intermédio de observações humanas, inspeções, valores característicos heurísticos tais como tipos de ruídos, cores, vibrações, etc. Já o diagnóstico de falhas consiste em determinar o tipo, tamanho e lugar da falha baseado nos sintomas analíticos e heurísticos observados (RIVIERA, 2007).

Diferentemente destas metodologias baseadas em bases de conhecimento, métodos de detecção de falhas baseados em modelos matemáticos também têm sido desenvolvidos nas últimas três décadas, dentre os quais podemos citar os propostos por Himmelblau (1978), Iserman (1997), Sampath et al. (1995). Nos trabalhos de Iserman (1997) e Himmelblau (1978) são descritos os principais métodos de detecção e diagnóstico de falhas baseados em modelos, dos quais se destacam:

i) Detecção de falhas com estimação de parâmetros e observadores (Filtro de Kalman).

ii) Diagnóstico de falhas usando métodos de classificação e reconhecimento de padrões, através de classificação geométrica e estatística.

iii) Reconhecimento de padrões com redes neurais e classificação com clusterização fuzzy (HALGAMUGE, 1996).

iv) Diagnóstico de falhas usando métodos de inferência com aproximações feitas por análise de árvores de falha (FTA, *fault-tree analysis*), análises de eventos falha (ETA, *event-tree analysis*)

v) Inteligência artificial, redes bayesianas e lógica Fuzzy (TEIXEIRA, 1993; KASZKUREWICZ et al., 1997).

vi) Diagnóstico de falhas com abordagem de sistemas híbridos (JIROVEANU et al., 2006).

vii) Diagnóstico de falhas usando Sistemas a Eventos Discretos (SAMPATH et al., 1995; LAFORTUNE et al., 2001).

No contexto de Sistemas a Eventos Discretos (SEDs) podemos destacar os trabalhos de Lin (1994), Sampath et al. (1995), Sampath et al. (1996), Debouk et al. (2000), Jiang et al. (2001), Lafortune et al. (2001), Zad et al. (2003), Qiu e Kumar (2005), Wang et al. (2007), Basílio e Lafortune (2009), Basílio et al. (2010), Athanasopoulou et al. (2010), Carvalho et al. (2010), Carvalho et al. (2013) e Lin et al. (2013). O trabalho apresentado por Lin et al.(1994) trouxe pela primeira vez a temática de diagnóstico de falhas no contexto de Sistemas a Eventos Discretos, onde os conceitos sobre capacidade de diagnosticar uma falha no sistema foram introduzidos. Posteriormente nos trabalhos de Sampath et al. (1995) e Sampath et al. (1996), os autores apresentam uma metodologia para a modelagem de sistemas físicos como sistema a eventos discretos, introduzem o conceito de mapeamento de sensores, apresentam a definição

de diagnosticabilidade de linguagens geradas por SEDs e um procedimento para a construção de diagnosticadores de falhas e, por fim, dão as condições necessárias e suficientes para diagnosticabilidade (RIVIERA, 2007). O diagnóstico de falhas estima a ocorrência da falha a partir de sequências de eventos parcialmente observadas, através de observadores modificados denominados diagnosticadores. Ainda nos trabalhos de Sampath et al. (1995; 1996), para diagnóstico de falhas em SEDs o diagnosticador é proposto com duas finalidades (MOREIRA et al., 2010):

i) detecção *online* e isolação de falhas de um sistema e;

ii) verificação *offline* das propriedades de diagnosticabilidade de um sistema.

Outros trabalhos usando sistemas modelados por autômatos foram feitos por Lunze e Schröder (2004), onde visualizaram o problema de diagnóstico como um problema de observação, por intermédio de uma abordagem híbrida usando autômatos estocásticos (RIVIERA, 2007). Um autômato estocástico é um autômato ao qual é adicionada uma estrutura probabilística para estimar a probabilidade de ocorrência de eventos específicos (BASILIO, CARVALHO e MOREIRA, 2010).

De acordo com Basílio et al. (2010), as metodologias desenvolvidas para o diagnóstico de falhas em SEDs podem ser aplicadas não apenas para sistemas onde o modelo de eventos discretos é o mais adequado (como redes de comunicação, sistemas computacionais e sistemas de produção), mas também em muitos sistemas dinâmicos de variáveis contínuas, uma vez que esses sistemas, considerando um nível maior de abstração, também podem ser modelados como SEDs, o que torna essa abordagem ainda mais ampla e relevante.

Uma vez que a ocorrência da falha não pode ser registada pelos sensores deve-se considerá-la como um evento não observável no modelo do sistema. O diagnosticador é um autômato, onde cada estado é constituído por um conjunto de estados do sistema que representam uma

estimativa do estado atual do sistema baseado apenas na ocorrência dos eventos observáveis (que podem registrados por sensores) e têm marcações que indicam se a sequência de eventos observáveis que ocorreram tem ou não o evento de falha (LIMA, 2010). Assim em um sistema físico modelado como SED, se presume que os sensores sempre poderão informar a ocorrência dos eventos associados a eles. Porém, um mau funcionamento dos sensores pode causar a perda da observabilidade desses eventos, o que prejudicaria o diagnóstico de falhas. Rohloff (2005) aborda o problema de controle tolerante a falha e propõe um teste para verificar a existência de um controle supervisório (RAMADGE e WONHAM, 1989) para o caso de sistemas sujeitos a falhas permanentes de sensores. Lima et al. (2010a) também abordam este problema e propõem um diagnosticador robusto à perda permanente de sensores e ainda Carvalho et al. (2010) propõem duas estratégias sistemáticas para diagnóstico de falhas em sistemas sujeitos a perda intermitente de sensores. Uma vez que o diagnóstico de falhas é feito considerando apenas os eventos observáveis, seria possível determinar um conjunto mínimo destes eventos que seriam realmente relevantes para a realização do diagnóstico? Uma solução para esta questão é apresentada nos trabalhos de Lima et al. (2010b) e Basilio et al. (2012), onde em ambos são apresentados os fundamentos teóricos necessários para o desenvolvimento de um algoritmo de busca capaz de encontrar todos os subconjuntos do conjunto de eventos observáveis (denominada base para diagnóstico), que são necessários para diagnosticar a ocorrência da falha.

Esta dissertação fornece uma base teórica para o estudo e pesquisa sobre diagnóstico de falhas em Sistemas a Eventos Discretos Modelados por autômatos, tanto para o diagnóstico centralizado como o diagnóstico descentralizado com a coordenação (co-diagnose). A principal contribuição deste trabalho é a proposta de uma arquitetura para diagnóstico de falhas em sistemas embarcados modelados a eventos discretos. Nesta arquitetura considera-se o sistema dividido em camadas, para as quais os diagnosticadores são especificamente projetados a fim de obter um diagnóstico mais preciso, que possibilite uma ação de recuperação ou

isolação da falha mais eficaz, quando necessário. Para tanto, são consideradas as seguintes camadas:

i) camada de *Software*: Relacionada às rotinas de *software* (controle e sistema)

ii) camada de *Hardware*: Relacionada aos circuitos e componentes da placa eletrônica

iii) camada de Cargas: Relacionada às cargas e/ou atuadores externos

A segunda contribuição deste trabalho é uma proposta de implementação em *software* desses diagnosticadores, utilizando a linguagem C ANSI (KERNIGHAN e RITCHIE, 1988) e também o desenvolvimento de uma ferramenta para a geração automática deste *software*.

Uma terceira contribuição deste trabalho é um estudo de caso para um refrigerador *Frost Free*, onde diagnosticadores para as três camadas (*software*, *hardware* e cargas) foram modelados, implementados em *software* e embarcados no microcontrolador da placa de controle do refrigerador.

Este trabalho está estruturado da seguinte maneira. O capítulo 2 apresenta uma revisão teórica dos conceitos relevantes para o estudo de diagnóstico de falhas em SEDs, define os conceitos de diagnosticabilidade, apresenta as condições necessárias e suficientes para o diagnóstico, apresenta a metodologia para construção de diagnosticadores tanto para o caso de diagnose centralizada como descentralizada (com coordenação). O Capítulo 3 apresenta a proposta de arquitetura de diagnóstico para sistemas embarcados onde os diagnosticadores projetados segundo a teoria apresentada no capítulo anterior podem ser empregados. O capítulo 4 mostra os detalhes do projeto dos diagnosticadores propostos para um caso real em um refrigerador *Frost Free*. Já o capítulo 5 mostra os detalhes da implementação em *software* dos diagnosticadores obtidos no capítulo anterior a apresenta a ferramenta criada para a geração automática deste *software*. Por fim, o capítulo 6 mostra como foi testado e

validado os diagnosticadores após serem embarcados no microcontrolador junto ao *software* principal de controle do refrigerador. Conclusões, discussões e propostas para trabalhos futuros são apresentados no capítulo 7.

2 FUNDAMENTOS TEÓRICOS DE DIAGNOSE DE FALHAS EM SEDS

O objetivo deste capítulo é dar subsídios para o entendimento dos formalismos teóricos aplicados nos capítulos seguintes. Para isso será feita uma breve revisão teórica dos conceitos fundamentais necessários para o estudo da diagnose de falhas em Sistemas a Eventos Discretos (SEDs), tais como linguagens e autômatos, autômato determinístico e não-determinístico, projeção e projeção inversa de linguagens, SEDs parcialmente observadas e observadores. Também serão revistos os principais conceitos e resultados mais relevantes da teoria de Sistemas a Eventos Discretos (SEDs) aplicados à diagnose de falhas.

2.1 Sistemas a Eventos Discretos

Denominam-se Sistemas a Eventos Discretos (SEDs), sistemas dinâmicos cujo estados podem ser representados de maneira discreta e a transição destes estados se dá através de eventos. Um estado discreto significa que o mesmo pode assumir valores simbólicos, como por exemplo: {ligado, desligado}, {aberto, fechado}, {alto, médio, baixo} ou valores numéricos, como por exemplo: {0, 1, 2, ...}. Já os eventos, geralmente estão associados a ações específicas, como por exemplo fechar uma chave, atingir um faixa de temperatura ou pode ainda ser o resultado de duas ou mais condições que são satisfeitas, como por exemplo uma chave ficou fechada por mais de um determinado tempo, uma temperatura não foi alcançada após um determinado tempo de uma resistência ligada, etc. Todas as sequências de eventos possíveis de serem geradas por um SED caracterizam a linguagem desse SED, sendo esta definida sobre o conjunto de eventos (alfabeto) do sistema. Embora alguns sistemas sejam naturalmente discretos e com a evolução determinada pela ocorrência de eventos, com um certo grau de abstração, é possível modelar qualquer sistema físico como um SED, desde que este modelo seja

capaz de reproduzir, dentro de limites de tolerância pré-estabelecidos, o comportamento do sistema (BASILIO et al. 2010).

2.2 Linguagens e Autômatos Determinísticos de Estados Finitos

2.2.1 Linguagens

Todo SED possui um conjunto de eventos Σ a ele associado. Esse conjunto pode ser interpretado como o *alfabeto* de uma linguagem, onde as possíveis sequências formadas por elementos desse conjunto podem ser interpretadas como *palavras* de uma linguagem sobre o alfabeto. Uma sequência que não possui eventos é formada pela sequência vazia ε. Assim, é possível utilizar linguagens para modelar o comportamento de um SED. Formalmente, o conceito de linguagem é definido como segue (HOPCROFT et al., 2007):

Definição 1. *(Linguagem)* Uma linguagem definida sobre um conjunto de eventos Σ é um conjunto formado por sequências de comprimento finito construídas a partir de eventos pertencentes a Σ.

Para ilustrar a definição acima, suponha o seguinte conjunto de eventos $\Sigma = \{a, b, c\}$. Pode-se definir a linguagem $L_1 = \{\varepsilon, ab, abab\}$, formada por três sequências apenas; a linguagem $L_2 = \{ca, cb, cc\}$, formada por todas as possíveis sequências de tamanho 2 que iniciam com o evento c.

Algumas operações podem ser definidas sobre o conjunto de eventos ou sobre sequências, uma delas é o *Fecho de Kleene* de um conjunto Σ, denotado por Σ^*, que define o conjunto de todas as sequências finitas formadas por elementos de Σ, incluindo a sequência vazia ε. Assim, Σ^* é um conjunto infinito, uma vez que contém sequências arbitrariamente longas. Se $\Sigma = \{a, b, c\}$ então $\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \ldots\}$. Formalmente, define-se o *Fecho de Kleene* como (CASSANDRAS e LAFORTUNE, 2008, p. 56):

Definição 2. *(Fecho de Kleene)* Seja $L \subseteq \Sigma^*$, então o *Fecho de Kleene* de L denotado por L^*, é definido como:

$$L^* := \{\varepsilon\} \cup L \cup LL \cup LLL \ldots \tag{2.1}$$

A operação de *concatenação*, consiste em unir duas ou mais sequências para formar uma nova sequência. Por exemplo, para o conjunto Σ anteriormente definido, a sequência ab é a concatenação dos eventos a e b. A sequência vazia ε é o elemento identidade da concatenação, sendo $s\varepsilon = \varepsilon s = s$ para qualquer sequência s. Formalmente o operação de concatenação é definida como segue (CASSANDRAS e LAFORTUNE, 2008, p. 56):

Definição 3. *(Concatenação)* Sejam $L_1, L_2 \subseteq \Sigma^*$, então a concatenação L_1L_2 é definida como:

$$L_1L_2 := \{s \in \Sigma^* : (s = s_1s_2)[s_1 \in L_1 \; e \; s_2 \in L_2]\} \tag{2.2}$$

Outras duas operações definidas sobre linguagens são as operações de *prefixo fechamento* e *pós linguagem*, cujas definições são apresentadas a seguir (CASSANDRAS e LAFORTUNE, 2008, p. 56):

Definição 4. *(Prefixo Fechamento)* Seja $L \subseteq \Sigma^*$, então o prefixo fechamento de L, denotado por $\overline{L}$ é definido como:

$$\overline{L} := \{s \in \Sigma^* : (\exists t \in \Sigma^*)[st \in L]\} \tag{2.3}$$

Se $L = \overline{L}$, diz-se que a linguagem L é prefixo-fechada.

Definição 5. *(Pós Linguagem)* Seja $L \subseteq \Sigma^*$ e $s \in L$, então a pós linguagem de L após s, denotada por L/s, é a linguagem:

$$L/s := \{t \in \Sigma^* : st \in L)\} \tag{2.4}$$

Outra operação bastante útil sobre linguagens e cujo conceito tem bastante relevância para o estudo de diagnose de falhas em SEDs é

a projeção de linguagens (RAMADGE e WONHAM, 1989; CASSANDRAS e LAFORTUNE, 2008). Quando aplicada a uma sequência de eventos, a projeção permite manter apenas o conjunto de eventos selecionados (eventos observáveis, por exemplo) desta sequência e apagar os demais (não observáveis, por exemplo). Formalmente é definida como:

Definição 6. *(Projeção)* Seja Σ e Σ_o conjuntos de eventos, onde $\Sigma_o \subset \Sigma$. A projeção P_o de uma sequência de eventos Σ em Σ_o é definida como:

$$P_o : \Sigma^* \to \Sigma_o^* \tag{2.5}$$

com as seguintes propriedades:

$$\begin{array}{rcl} P_o(\varepsilon) & := & \varepsilon, \\ P_o(\sigma) & := & \begin{cases} \sigma, & \text{se } \sigma \in \Sigma_o, \\ \varepsilon, & \text{se } \sigma \in \Sigma \backslash \Sigma_o, \end{cases} \\ P_o(s\sigma) & := & P_o(s)P_o(\sigma), \text{ para } s \in \Sigma^*, \text{ e } \sigma \in \Sigma \end{array} \tag{2.6}$$

onde $\backslash$ denota a diferença entre conjuntos. Assim a operação $\Sigma \backslash \Sigma_o = \Sigma - \Sigma_o$, isto é, o conjunto formado pelos elementos de Σ que não pertencem ao conjunto Σ_o.

O operador projeção pode ser estendido para uma linguagem L aplicando as definições mostradas acima a todas as sequências pertinentes à linguagem L (CARVALHO, 2011). Assim, se $L \subset \Sigma^*$ então:

$$P_o(L) := \{t \in \Sigma_o^* : (\exists s \in L)\,[P_o(s) = t]\} \tag{2.7}$$

O tipo de projeção apresentada na Definição 6 é denominada projeção natural. Pode-se também definir a projeção inversa P_o^{-1} de uma sequência, como segue:

Definição 7. *(Projeção Inversa)* A projeção inversa de uma sequência de eventos é definida como uma função

$$\begin{aligned} P_o^{-1} &: \Sigma_o^* \to 2^{\Sigma^*} \\ P_o^{-1}(t) &:= \{s \in \Sigma^* : P_o(s) = t\} \end{aligned} \tag{2.8}$$

Em outras palavras, a Definição 7 implica que para uma dada sequência formada pelos eventos pertencentes ao conjunto de menor cardinalidade Σ_o, a projeção inversa retornará o conjunto de todas as sequências formadas por eventos pertencentes ao conjunto de maior cardinalidade Σ, onde as projeções serão a própria sequência inicial (LIMA, 2010).

Assim como em (2.7), a projeção inversa também pode ser estendida para linguagens de modo que para $L \subset \Sigma_o^*$, tem-se:

$$P_o^{-1}(L_o) := \{s \in \Sigma^* : (\exists t \in L_o)[P_o(s) = t]\} \tag{2.9}$$

Para ilustrar o conceito de projeção, considere o exemplo a seguir.

Exemplo 1. Seja $\Sigma_s = \{a,b,c\}$, e seus respectivos subconjuntos $\Sigma_1 = \{a,b\}$ e $\Sigma_2 = \{a,c\}$ e também a linguagem $L = \{c, ccb, bca, cabbc, abcbca\} \subset \Sigma_s^*$. Para as duas projeções $P_i(L) : \Sigma_s^* \to \Sigma_i^*, i = 1,2$ tem-se que:

$$\begin{aligned} P_1(L) &= \{\varepsilon, b, ba, abb, abba\} \\ P_2(L) &= \{c, cc, ca, cac, acca\} \\ P_1^{-1}(\{\varepsilon\}) &= \{c\}^* \\ P_1^{-1}(\{b\}) &= \{c\}^* \{b\} \{c\}^* \\ P_1^{-1}(\{a,b\}) &= \{c\}^* \{a\} \{c\}^* \{b\} \{c\}^* \end{aligned}$$

Dada a importância das projeções naturais no estudo de diagnose de falhas de SEDs, seguem algumas propriedades relevantes que auxiliarão o entendimento deste trabalho.

Propriedades de Projeções Naturais (CASSANDRAS e LAFORTUNE, 2008, p. 58):

1. $P[P^{-1}(L)] = L$
   $L \subseteq P^{-1}[P(L)]$
2. Se $A \subseteq B$ então $P(A) \subseteq P(B)$ e $P^{-1}(A) \subseteq P^{-1}(B)$
3. $P(A \cup B) = P(A) \cup P(B)$
   $P(A \cap B) \subseteq P(A) \cap P(B)$
4. $P^{-1}(A \cup B) = P^{-1}(A) \cup P^{-1}(B)$
   $P^{-1}(A \cap B) = P^{-1}(A) \cap P^{-1}(B)$
5. $P(AB) = P(A)P(B)$
   $P^{-1}(AB) = P^{-1}(A)P^{-1}(B)$

2.2.2 Autômatos Determinísticos de Estados Finitos

Neste trabalho os SEDs considerados serão modelados usando autômatos de estados finitos, também chamados de máquinas de estados de estados finitos. Por simplicidade, ao longo do trabalho utiliza-se apenas o termo autômato para se referir a autômato de estados finitos.

Um autômato é um dispositivo capaz de representar uma linguagem através de regras bem definidas. Segundo Cassandras e Lafortune (2008) um autômato determinístico é formalmente definido como segue:

Definição 8. *(Autômato Determinístico)* Um autômato determinístico, denotado por G, é definido pela seguinte sêxtupla:

$$G = (X, \Sigma, f, \Gamma, x_0, X_m), \tag{2.10}$$

no qual:

- X denota o espaço de estados;
- Σ é o conjunto de eventos;
- $f : X \times \Sigma \to X$ é a função de transição de estados;
- $\Gamma : X \to 2^{\Sigma}$ é a função dos eventos ativos;
- x_0 é o estado inicial;
- $Xm \subseteq X$ é o conjunto de estados marcados

Diz-se que o autômato é determinístico pois f é uma função de $X \times \Sigma$ para X, ou seja, a ocorrência de um evento leva o autômato de seu estado atual para um único estado futuro. Em outras palavras, não é possível haver mais do que uma transição rotulada pelo mesmo evento saindo de um mesmo estado do autômato. Já em um autômato não-determinístico, f é uma função de $X \times \Sigma$ para 2^X. Nesse caso, podem haver diversas transições rotuladas pelo mesmo evento saindo de um mesmo estado, isto é, um mesmo evento pode levar o autômato de seu estado atual para mais de um estado futuro (CASSANDRAS e LAFORTUNE, 2008). O conceito de autômato não-determinístico será detalhado posteriormente na seção 2.3.

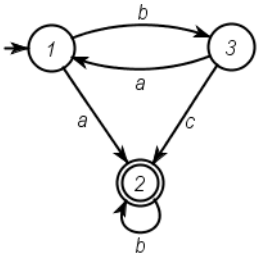
Para o conjunto de eventos Σ, Σ^* denota o conjunto de todas as sequências possíveis de comprimento finito com elementos pertencentes a Σ, incluindo a sequência vazia ε. Por conveniência, f é sempre estendida do domínio $X \times \Sigma$ para o domínio $X \times \Sigma^*$, da seguinte maneira recursiva (CASSANDRAS e LAFORTUNE, 2008):

$$\begin{aligned} f(x,\varepsilon) &:= x \\ f(x,s\sigma) &:= f[f(x,s),\sigma] \text{ para } s \in \Sigma^* \text{ e } \sigma \in \Sigma \end{aligned} \tag{2.11}$$

O autômato tal como descrito na Definição 8 pode ser representado graficamente por meio de um diagrama de transição de estados, onde os estados são representados por nós circulares, as transições por arcos rotulados pelos eventos. O estado inicial é identificado por uma seta

e os estados marcados são representados por círculos duplos concêntricos. A Figura 2.1 mostra um exemplo de um autômato onde é possível verificar os conceitos acima descritos.

Figura 2.1: Exemplo de um Diagrama de Transição de Estados para um Autômato



Fonte: Autor

O grafo da Figura 2.1, representa o diagrama de transição de estados de um autômato do tipo $G = (X, \Sigma, f, \Gamma, x_0, X_m)$, onde:

- $X = \{1,2,3\}$;
- $\Sigma = \{a,b,c\}$;
- $Xm = \{2\}$;
- $x_0 = 1$;
- $f(1,a) = 2$, $f(1,b) = 3$, $f(3,a) = 1$, $f(3,c) = 2$, $f(2,b) = 2$;
- $\Gamma(1) = \{a,b\}$, $\Gamma(2) = \{b\}$, $\Gamma(3) = \{a,c\}$

Quando um evento ocorre em um autômato e o estado destino é o próprio estado de origem, ou seja, o evento não gera a mudança de estado, diz-se que o estado possui um auto-laço. Esta condição pode ser vista do estado 2 do autômato da Figura 2.1. Por outro lado, quando um estado não possui nenhum evento ativo, o denominamos como um estado de bloqueio ou *deadlock*.

A relação entre linguagens e autômatos pode facilmente ser identificada analisando-se o diagrama de transição de estados de um autômato. O conjunto de todas as sequências de eventos possíveis de serem executadas a partir do estado inicial forma a linguagem gerada por um autômato. Já o conjunto de sequências pertencentes à linguagem gerada que levam o sistema a um estado marcado constitui a linguagem marcada por um autômato. Normalmente, a linguagem marcada está relacionada com a linguagem de interesse de um autômato, podendo representar, por exemplo, o fim de uma tarefa, ou a disponibilidade de um recurso do sistema. As definições formais são apresentadas a seguir (CASSANDRAS e LAFORTUNE, 2008):

Definição 9. *(Linguagens gerada e marcada)* A linguagem gerada por $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ é definida como:

$$L(G) := \{s \in \Sigma^* : f(x_0, s) \text{ é definida}\}. \tag{2.12}$$

A linguagem marcada por G é definida como:

$$L_m(G) := \{s \in L(G) : f(x_0, s) \in X_m\}. \tag{2.13}$$

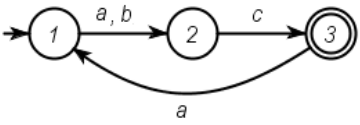
O exemplo a seguir ilustra a determinação das linguagens gerada e marcada por um autômato.

Exemplo 2. Seja o autômato da Figura 2.2, onde $\Sigma = \{a, b, c\}$. A linguagem gerada por G é $L(G) = \overline{\{\{a,b\}\{c\}\{a\}\}^*}$, enquanto que a linguagem marcada por G é $L_m(G) = \{\{a,b\}\{c\}\{\{a\}\{a,b\}\{c\}\}^*\}$

2.2.3 Operações com Autômatos

Algumas operações com autômatos são definidas para que se possa modificar, combinar ou compor dois ou mais autômatos, para assim se obter uma melhor representação de um sistema completo a partir de modelos de componentes individuais do sistema. Algumas destas principais operações que são relevantes para o estudo de diagnose de falhas

Figura 2.2: Autômato para o Exemplo 2



Fonte: Autor

em sistemas a eventos discretos são apresentadas a seguir (CASSANDRAS e LAFORTUNE, 2008).

Definição 10. *(Parte Acessível)* A parte acessível de um autômato G é a operação unária que elimina todos os estados de G que não são alcançáveis a partir do estado inicial x_0, assim como as transições relacionadas com esses estados eliminados. Seja $G = (X, \Sigma, f, \Gamma, x_0, X_m)$, formalmente define-se a parte acessível de G, denotada por $Ac(G)$ como o subautômato:

$$Ac(G) = \{X_{ac}, \Sigma, f_{ac}, x_0, X_{ac,m}\}. \tag{2.14}$$

sendo,

$$\begin{array}{rcl} X_{ac} & = & \{x \in X : (\exists s \in \Sigma^*)[f(x_0, s) = x]\}; \\ X_{ac,m} & = & X_m \cap X_{ac}; \\ f_{ac} & : & X_{ac} \times \Sigma \to X_{ac}; \end{array} \tag{2.15}$$

onde f_{ac} denota a nova função de transição obtida restringindo-se o domínio de f para o domínio dos estados acessíveis X_{ac}.

Definição 11. *(Parte Coacessível)* Parte coacessível de um autômato G é uma operação unária e é obtida eliminando-se todos os estados de G a partir dos quais não é possível alcançar um estado marcado. Seja $G = (X, \Sigma, f, \Gamma, x_0, X_m)$. Formalmente define-se a parte coacessível de G,

denotada por $CoAc(G)$ como o subautômato*:*

$$CoAc(G) = \{X_{coac}, \Sigma, f_{coac}, x_{0,coac}, X_m\}. \tag{2.16}$$

sendo,

$$\begin{aligned} X_{coac} &= \{x \in X : (\exists s \in \Sigma^*)[f(x,s) \in X_m]\} \\ x_{0,coac} &= \begin{cases} x_0, & \textit{se } x_0 \in X_{coac}, \\ \textit{indefinido}, & \textit{se } x_0 \notin X_{coac} \end{cases} \\ f_{coac} &: X_{coac} \times \Sigma \to X_{coac} \end{aligned}$$

onde f_{coac} denota a nova função de transição obtida restringindo-se o domínio de f aos estados coacessíveis X_{coac}.

Além das duas operações unárias acima descritas, é possível fazer a composição de dois ou mais autômatos e assim obter um novo autômato. Para isso, pode-se usar as operações de produto ou composição paralela definidas a seguir (CASSANDRAS e LAFORTUNE, 2008).

Definição 12. *(Produto)* Sejam os autômatos $G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ e $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0_2}, X_{m_2})$. Então, o produto de G_1 e G_2, denotado por $G_1 \times G_2$ é formalmente definido como*:*

$$G_1 \times G_2 = Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f_{1\times 2}, (x_{0_1}, x_{0_2}), X_{m_1} \times X_{m_2}) \tag{2.17}$$

sendo,

$$f_{1\times 2}((x_1,x_2),\sigma) = \begin{cases} (f_1(x_1,\sigma), f_2(x_2,\sigma)), \textit{ se } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2); \\ \textit{indefinido, caso contrário} \end{cases}$$

A definição acima implica que qualquer transição de estados no produto $G_1 \times G_2$ ocorre se, e somente se, essa transição é possível em ambos autômatos G_1 e G_2. Ou seja, a evolução de estados em $G_1 \times G_2$ é completamente sincronizada com a evolução de estados dos autômatos G_1 e G_2.

Conforme visto acima a composição por produto é restritiva, pois só permite transições em eventos comuns. Contudo, essa operação não é

muito adequada quando se modela sistemas compostos por componentes que interagem entre si, onde o conjunto de eventos de cada componente inclui eventos particulares (que pertencem ao seu próprio comportamento interno) e eventos comuns que são compartilhados com outros autômatos e captam a conexão entre os respectivos componentes do sistema. A forma adequada para construção de modelos de sistemas inteiros a partir de modelos de componentes individuais do sistema é por meio da composição paralela (CASSANDRAS e LAFORTUNE, 2008), cuja definição é mostrada a seguir.

Definição 13. *(Composição Paralela)* Sejam os autômatos:
$G_1 = (X_1, \Sigma_1, f_1, \Gamma_1, x_{0_1}, X_{m_1})$ e $G_2 = (X_2, \Sigma_2, f_2, \Gamma_2, x_{0_2}, X_{m_2})$.
Então, a composição paralela de G_1 e G_2, denotada por $G_1 \parallel G_2$ é formalmente definido como*:*

$$G_1 \parallel G_2 = Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f_{1\|2}, (x_{0_1}, x_{0_2}), X_{m_1} \times X_{m_2}) \tag{2.18}$$

de modo que:

$$f_{1\|2} : (X_1 \times X_2) \times (\Sigma_1 \cup \Sigma_2) \rightarrow (X_1 \times X_2)$$

ou seja:

$$f_{1\|2}((x_1, x_2), \sigma) = \begin{cases} (f_1(x_1, \sigma), f_2(x_2, \sigma)), \textit{ se } \sigma \in \Sigma_1 \cap \Sigma_2 \textit{ e} \\ \sigma \in \Sigma_1(x_1) \cup \Sigma_2(x_2); \\ (f_1(x_1, \sigma), x_2), \textit{ se } \sigma \in \Sigma_1 \textit{ e } \sigma \notin \Sigma_2 e \sigma \in \Sigma_1(x_1); \\ (x_1, f_2(x_2, \sigma)), \textit{ se } \sigma \in \Sigma_2 \textit{ e } \sigma \notin \Sigma_1 e \sigma \in \Sigma_1(x_1); \\ \textit{indefinida, caso contrário} \end{cases}$$

Na composição paralela um evento comum, isto é, um evento em $\Sigma_1 \cap \Sigma_2$, apenas pode ser executado se os dois autômatos executá-lo simultaneamente. Assim, os dois autômatos estão sincronizados nos eventos comuns. Já os eventos particulares, isto é, aqueles onde $(\Sigma_2 \backslash \Sigma_1) \cup (\Sigma_1 \backslash \Sigma_2)$, não estão sujeitas a essa restrição e podem ser executados

sempre que possível. Neste tipo de interconexão, um componente pode executar seus eventos particulares sem a participação do outro componente, contudo um evento comum só pode acontecer se ambos os componentes podem executá-lo (CASSANDRAS e LAFORTUNE, 2008).

2.3 Autômatos Não-Determinísticos

No autômato determinístico definido na seção 2.2.2, o estado inicial é um estado único e um evento $\sigma \in \Sigma$ causa uma transição de x para um único estado $y = f(x, \sigma)$. Contudo, é possível que um evento σ no estado x cause uma transição para mais de um estado, isso pode ocorrer por exemplo devido à falta de informação do sistema modelado, onde o efeito de um evento não é perfeitamente conhecido, não sendo possível então precisar o estado futuro. Outra possibilidade, é a necessidade de se agrupar alguns estados do autômato o que resultaria em múltiplas transições todas com o mesmo rótulo saindo do mesmo estado (agora agrupado em um só). Nesse caso, $f(x, \sigma)$ não representará mais um único estado mas sim um conjunto de estados. Tem-se ainda a possibilidade de ε estar presente no diagrama de transição de estados de um autômato, ou seja, alguns estados podem estar conectados por transições rotuladas pela sequência vazia denominadas de *transições-ε*. Essas transições podem representar, por exemplo, eventos que causam mudanças no estado de um SED, porém a ocorrência de tais eventos não são percebidos por um observador externo. Isso pode ocorrer por exemplo devido à falta de sensores capazes de registrar esses eventos (CASSANDRAS e LAFORTUNE, 2008). Considerando essas possibilidades, define-se uma classe denominada autômato não-determinístico, cuja representação formal é apresentada a seguir (CASSANDRAS e LAFORTUNE, 2008, p.70):

Definição 14. *(Autômato Não-Determinístico):* Um autômato não-determinístico é definido pela sêxtupla:

$$G_{nd} = (X, \Sigma \bigcup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m),$$

em que os parâmetros possuem a mesma interpretação dada na definição do autômato determinístico, com apenas duas ressalvas:

i) f_{nd} é uma função $f_{nd} : X \times \Sigma \cup \{\varepsilon\} \rightarrow 2^X$, ou seja, $f_{nd}(x,\sigma) \subseteq X$ sempre que for definida, sendo 2^X o conjunto de subconjuntos de X, também denominado conjunto das partes de X (CURY, 2001).

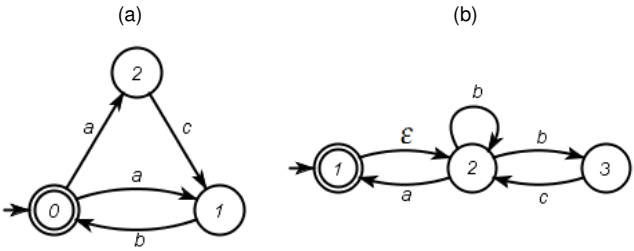
ii) O estado inicial pode ser um conjunto de estados, ou seja, $x_o \subseteq X$.

Para ilustrar esses conceitos, seguem dois exemplos de autômatos não-determinísticos.

Exemplo 3. *Autômato não-determinístico simples:* Considere o autômato mostrado na Figura 2.3a. Quando o evento a ocorre no estado 0, existem duas transições possíveis, para o estado 1 e para o estado 2. Assim, após a ocorrência do evento a não é possível determinar com exatidão qual o estado atual do autômato. As transições de estados mapeadas para esse autômato são: $f_{nd}(0,a) = \{1,2\}$, $f_{nd}(1,b) = \{0\}$ e $f_{nd}(2,c) = \{1\}$, onde os valores de f_{nd} são expressados como subconjuntos do conjunto de estados X. As transições $f_{nd}(0,b)$, $f_{nd}(1,a)$ e $f_{nd}(2,a)$ não são definidas.

Exemplo 4. *Autômato não-determisnístico com transição-ε:* Considere agora o autômato mostrado na Figura 2.3b que inclui uma *transição-ε*. Temos que: $f_{nd}(1,\varepsilon) = \{2\}$, $f_{nd}(2,b) = \{2,3\}$, $f_{nd}(2,a) = \{1\}$ e $f_{nd}(3,c) = \{2\}$. Como se pode observar, a ocorrência do evento b no estado 2 pode levar o autômato tanto para o estado 3 como fazê-lo permanecer no próprio estado 2. Suponha que imediatamente após se ligar o sistema se observe a ocorrência do evento a, a transição $f_{nd}(1,a)$ não é definida, assim se pode concluir que ocorreu uma transição do estado 1 para o estado 2 seguida do evento a, fazendo o sistema retornar para o estado 1. A transição do estado 1 para o estado 2, sem que esta tenha sido observada é devida à *transição-ε*

Figura 2.3: (a) Autômato não-determinístico do exemplo 3; (b)Autômato não-determinístico com *transição-ε* do exemplo 4



Fonte: Autor

Pode-se estender f_{nd} para uma sequência u, em vez de aplicá-la apenas a um único evento, como foi feito para f no caso de autômatos determinísticos. Em particular:

$$f_{nd}(x,u\sigma) := \{z : z \in f_{nd}(y,\sigma) \textit{ para algum estado } y \in f_{nd}(x,u)\} \quad (2.19)$$

Ou seja, após a ocorrência da sequência u, todos os estados y que são acessíveis a partir de x são identificados e assim, pela ocorrência do evento σ, chega-se ao estado z que pertence ao conjunto $f_{nd}(x,u\sigma)$.

A linguagem que pode ser gerada e marcada por um autômato não-determinístico é definida como (CASSANDRAS e LAFORTUNE, 2008):

$$L(G_{nd}) = \{s \in \Sigma^* : (\exists x \in x_0)[f_{nd}(x,s) \text{ é definida}]\}$$
$$L_m(G_{nd}) = \{s \in L(G_{nd}) : (\exists x \in x_0)[f_{nd}(x,s) \cap X_m \neq \emptyset\}$$

Conforme Cassandras e Lafortune (2008, p.72), as definições acima possuem o seguinte significado: Uma sequência pertence a uma

linguagem gerada por um autômato não-determinístico se no diagrama de transição de estados existe um caminho que é consistentemente rotulado por esta sequência. Se é possível seguir este caminho e este termina em um estado marcado, então essa sequência está contida na linguagem marcada pelo autômato.

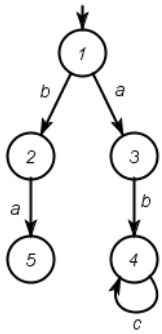
2.4 SEDs Parcialmente Observados

Conforme mencionado anteriormente, em um autômato não- determinístico as transições definidas por ε representam eventos que ocorrem no sistema modelado mas que não podem ser registrados ou observados. Essa falta de observabilidade pode ocorrer, por exemplo, devido à falta de sensores capazes de registrar tais eventos ou por estes eventos ocorrerem em uma parte remota que não se comunica com o sistema modelado.

Se considerarmos o conjunto de eventos Σ composto por dois subconjuntos disjuntos, Σ_o referente ao conjunto de eventos observáveis e Σ_{uo}, referente ao conjunto de eventos não observáveis, isto é $\Sigma = \Sigma_o \cup \Sigma_{uo}$, então em um autômato onde $\Sigma_{uo} = \emptyset$, ou seja, o conjunto de eventos Σ é composto apenas por eventos observáveis (Σ_o), a ocorrência de um evento σ causa a transição do autômato de um estado x conhecido para um estado y também conhecido. No entanto, quando consideramos $\Sigma_{uo} \neq \emptyset$, isto é, o conjunto Σ agora possui eventos não observáveis, teremos uma indeterminação quanto ao estado atual do autômato, devido à não observação dos eventos pertencentes ao subconjunto Σ_{uo}. Na Figura 2.4 é mostrado um autômato que ilustra essa possibilidade. Considerando que o evento a não pode ser observado, então, após a ocorrência (observação) do evento b não será possível determinar com exatidão o estado atual do sistema (autômato), uma vez que haverá três estados possíveis, os estados 2, 4 e 5.

Neste contexto, eventos de falha que não causam mudanças nos sensores podem ser também considerados como eventos não observáveis. Então, se ao invés de simplesmente rotularmos esses eventos

Figura 2.4: Autômato determinístico para o evento a não observável



Fonte: Autor

não observáveis como sequências vazias ε e obter um autômato não-determinístico, rotularmos essas transições como eventos genuínos, porém classificar esses eventos como não observáveis, obteremos assim um autômato determinístico, onde o conjunto de eventos Σ é composto pelos subconjuntos Σ_o e Σ_{uo}. Nestas condições, o SED pode ser denominado como "parcialmente observado"(CASSANDRAS e LAFORTUNE, 2008, p.102).

Um autômato determinístico com eventos não observáveis que representa um SED parcialmente observado, é definido como:

$$G = (X, \Sigma, f, \Gamma, x_0, X_m), \tag{2.20}$$

onde, $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$, sendo $\dot{\cup}$ a partição de um conjunto, i.e. $\Sigma_o \cap \Sigma_{uo} = \emptyset$ e $\Sigma_o \cup \Sigma_{uo} = \Sigma$.

2.5 Observador

O autômato determinístico que descreve o comportamento dinâmico de um autômato determinístico com eventos não observáveis, mencionado anteriormente, é chamado observador. O conjunto de eventos do observador é formado apenas pelos eventos observáveis e seu estado atual representa uma estimativa do estado atual do autômato parcialmente observado, atualizado após a ocorrência de um evento observável. O observador de G, denotado como $Obs(G)$ é definido da seguinte forma:

$$Obs(G) = (X_{obs}, \Sigma_o, f_{obs}, \Gamma_{obs}, x_{0obs}, X_{mobs}), \tag{2.21}$$

onde $X_{obs} \in 2^X$ e $X_{mobs} = \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$.

Um algoritmo para a construção de observadores para autômatos parcialmente observados, proposto por Basilio et al. (2010, p. 517) é apresentado a seguir. Contudo, esse algoritmo requer a noção de alcance não observável, denotado por $UR(x)$, cuja definição é apresentada a seguir (CASSANDRAS e LAFORTUNE, 2008, p.102):

Seja $G = (X, \Sigma_o \cup \Sigma_{uo}, f, x_0, X_m)$ um autômato parcialmente observado. Para cada estado $x \in X$ define-se:

$$UR(x) := \{y \in X : (\exists t \in \Sigma_{uo}^*)[f(x,t) = y]\}$$

A definição acima é estendida para um conjunto de estados $B \subseteq X$, da seguinte maneira:

$$UR(B) = \bigcup_{x \in B} UR(x)$$

Então $Obs(G) = (X_{obs}, \Sigma_o, f_{obs}, x_{0obs}, X_{m,obs})$ pode ser construído através do algoritmo a seguir (BASILIO et al., 2010, p. 517):

Algoritmo 1. (*Construção de Observadores*):

Passo 1: Defina $x_{0obs} = UR(x_0)$ e faça: $X_{obs} = \{x_{obs}\}$ e $\tilde{X}_{obs} = X_{obs}$
Passo 2: $\hat{X}_{obs} = \tilde{X}_{obs}$ e $\tilde{X}_{obs} = \emptyset$
Passo 3: Para cada $B \in \hat{X}_{obs}$
• $\Gamma_{obs}(B) = (\bigcup_{x \in B} \Gamma(x)) \cap \Sigma_o$,
• Para cada $e \in \Gamma_{obs}(B)$,
• $f_{obs}(B,e) = UR(\{x \in X : (\exists y \in B)[x = f(y,e)]\})$;
• $\tilde{X}_{obs} \leftarrow \tilde{X}_{obs} \cup f_{obs}(B,e)$
Passo 4: $X_{obs} \leftarrow X_{obs} \cup \tilde{X}_{obs}$
Passo 5: Repita os passos 2 a 4 até que toda a parte acessível de $Obs(G)$ tenha sido construída.
Passo 6: $X_{mobs} = \{B \in X_{obs} : B \cap X_m \neq \emptyset\}$

O objetivo desse algoritmo é calcular o alcance não-observável para cada estado de *G* alcançado por um evento observável. Para isso, calcula-se no primeiro passo o alcance não-observável do estado inicial x_0 para formar então o primeiro estado do observador. No passo 3, são calculados os conjuntos de eventos ativos dos estados do observador obtidos no passo ou na iteração anterior (sendo que o primeiro se refere ao alcance observável do estado inicial e o último aos estados de *G* alcançados por meio de eventos observáveis juntamente com os respectivos alcances não-observáveis). Posteriormente são calculados os próximos estados do observador, que correspondem aos alcances não observáveis dos estados de *G* alcançados a partir do estado atual do observador por meio de eventos observáveis. Essa sequência é repetida até que todos os estados acessíveis do observador tenham sido encontrados (BASILIO et al., 2010).

A partir do Algoritmo 1 e da definição de projeção de linguagens da Equação 2.7, pode-se concluir que $L(Obs(G)) = P_o[L(G)]$, ou, em outras palavras, a linguagem gerada por $Obs(G)$ é a projeção da linguagem de G sobre um conjunto de eventos observáveis (BASILIO et al., 2010).

Exemplo 5. Para melhor ilustrar a construção de observadores, consi-

dere o autômato parcialmente observado da Figura 2.5a, com um conjunto de eventos $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$ definido por: $\Sigma = \{a,b,c,d,\sigma_f,\sigma\}$, $\Sigma_o = \{a,b,c,d\}$, e $\Sigma_{uo} = \{\sigma_f,\sigma\}$ Aplicando o algoritmo 1, obtém-se o observador $G_{obs} = Obs(G)$, mostrado na Figura 2.5b, como segue:

i) $x_{0obs} = UR(x_0) = \{1\}$, $X_{obs} = \{\{1\}\}$ e $\tilde{X}_{obs} = \{\{1\}\}$

ii) $\hat{X}_{obs} = \{\{1\}\}$ e $\tilde{X}_{obs} = \emptyset$

iii) $\Gamma_{obs}(\{1\}) = \{a,b\}$,
$f_{obs}(\{1\},a) = UR(\{4\}) = \{4,3\}$,
$f_{obs}(\{1\},b) = UR(\{2\}) = \{4,2,3\}$, e $\tilde{X}_{obs} = \{\{4,3\},\{4,2,3\}\}$;

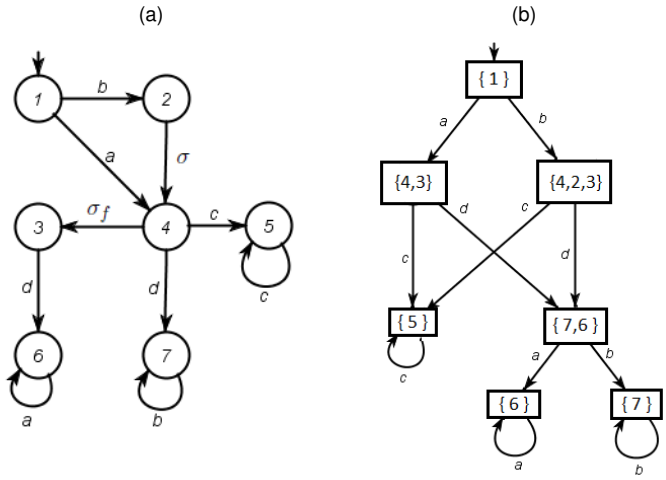
iv) $X_{obs} = \{\{1\},\{4,3\},\{4,2,3\}\}$;

Retorne ao passo 2 até que toda parte acessível de G_{obs} tenha sido construída.

2.6 Diagnosticabilidade

O conceito de diagnosticabilidade refere-se à capacidade de detectar qualquer falha em um sistema com um atraso finito, com base apenas na ocorrência de eventos observáveis registradas pelos sensores. Em relação a SEDs modelados por autômatos, de acordo com Basilio et al. (2010, p. 519), diz-se que uma linguagem gerada por um autômato é diagnosticável em relação a um conjunto de eventos observáveis Σ_o e a um conjunto de eventos de falha Σ_f, para $\Sigma_f \subseteq \Sigma_{uo}$ se a ocorrência da falha puder ser detectada após um número finito de transições depois da sua ocorrência, com base apenas em sequências de eventos observáveis. Em geral particiona-se o conjunto Σ_f em diferentes subconjuntos, isto é, $\Sigma_{f_i}, i = 1, 2, \ldots, m$, onde cada conjunto Σ_{f_i} é composto por eventos que modelam as falhas. Supondo que $\Pi_f = \{\Sigma_{f_1}, \Sigma_{f_2}, \ldots, \Sigma_{f_m}\}$ denote essa partição, então a ocorrência de uma falha f_i significa então a ocorrência de algum evento do conjunto Σ_{f_i}.

Figura 2.5: Exemplo 5: (a) Autômato Parcialmente Observado; (b) Observador



Fonte: Autor

Desde os primeiros estudos envolvendo diagnose de falhas em SEDs, as seguintes hipóteses são consideradas:

A1. (SAMPATH et al., 1995, p.1556) A linguagem L gerada por G é viva, ou seja, há uma transição definida para cada estado x em X, isto é, o sistema não pode alcançar um ponto onde nenhum evento é possível. Formalmente diz-se $\Gamma(x_i) \neq \emptyset$ para todo $x_i \in X$;

A2. (SAMPATH et al., 1995, p.1556) No autômato G não existe nenhum ciclo formado somente por eventos não observáveis, *i.e.* $\forall ust \in L, s \in \Sigma_{uo}^*, \exists n_0 \in \mathbb{N}$ tal que $\| s \| \leq n_0$, onde $\| s \|$ denota o comprimento da sequência s.

A3. (BASILIO e LAFORTUNE, 2009) Existe somente um tipo de falha, *i.e.*, $\Pi_f = \{\Sigma_f\}$, onde $\Sigma_f = \{\sigma_f\}$.

Na hipótese A1 considera-se que o sistema está sempre em operação. A hipótese A2 é necessária para evitar que, após a ocorrência do evento de falha, o mesmo possa se tornar indetectável se o sistema ficar preso em um ciclo de estados formados apenas por eventos não observáveis. No entanto, no trabalho apresentado por Basílio e Lafortune (2009), esta hipótese foi removida e tais ciclos referido como ciclos escondidos (BASILIO et al., 2010). Já a hipótese A3 é feita em alguns trabalhos apenas por simplicidade, uma vez que, para cada conjunto de eventos de falha do mesmo tipo um rótulo diferente deve ser usado, porém os fundamentos de diagnosticabilidade permanecem os mesmos utilizados para apenas um tipo de falha (BASILIO et al., 2010).

No trabalho de Sampath et al.(1995) as seguintes definições são apresentadas:

• $\Psi(\Sigma_{f_i}) = \{s\sigma_f \in L : \sigma_f \in \Sigma_{f_i}\}$: conjunto de todas as sequências de L que terminam com um evento de falha pertencente à Σ_{f_i}.

• $L/s = \{t \in \Sigma^* : st \in L\}$: continuação da linguagem L após a sequência s.

• $P_{oL}^{-1}(y) = \{s \in L : P_o(s) = y\}$: operador projeção inversa em L.

• A sequência $s \in L$ é uma sequência que contém a falha se $\Sigma_f \in s$.

Considerando $\sigma \in \Sigma$ e $s \in \Sigma^*$, usamos a notação $\sigma \in s$ para denotar que σ é um evento da sequência s. Com um ligeiro abuso de notação, podemos escrever que $\Sigma_{f_i} \in s$ para denotar o fato que $\sigma_f \in s$ para algum $\sigma_f \in \Sigma_{f_i}$ (SAMPATH et al., 1995) ou formalmente:

$$\overline{s} \cap \Psi(\Sigma_{f_i}) \neq \emptyset,$$

onde $\overline{s}$ denota o prefixo fechamento de s.

Finalmente após essas definições, pode-se formalmente apresentar a definição de diagnosticabilidade de uma linguagem, conforme descrito por Sampath et al. (1995):

Definição 15. Seja L a linguagem gerada por um autômato G e suponha que esta seja viva e prefixo-fechada. Então, L é diagnosticável em relação à projeção P_o e $\Sigma_f = \sigma_f$ se a seguinte condição for verdadeira:

$$(\exists n \in \mathbb{N})(\forall s \in \Psi(\Sigma_f))(\forall t \in L/s)(\| t \| \geq n \Rightarrow D),$$

e a condição de diagnose D expressa por:

$$(\forall \omega \in P_{oL}^{-1}(P_o(st)))(\Sigma_f \in \omega)$$

em que:

$\| t \|$ denota o comprimento da sequência t
s: sequência que termina com um evento de falha.
t: continuação de s

De acordo com Sampath et al. (1995), a definição de diagnosticabilidade acima tem o seguinte significado:

Seja s uma sequência arbitrária gerada pelo sistema, que termina com um evento de falha pertencente ao conjunto Σ_f e t qualquer subsequência que seja uma continuação da sequência s. A condição de diagnosticabilidade D requer que todas as sequências em L que gerem o mesmo registro de eventos observáveis, tais como a sequência st deve conter nela um evento de falha do conjunto Σ_f. Isto implica que em todas as continuações t de s é capaz detectar a ocorrência da falha com um atraso finito, mais especificamente, com um máximo de n transições do sistema após a sequência s. Em outras palavras, diagnosticabilidade implica que qualquer evento de falha deve levar a observações distintas

o bastante de tal modo a permitir a identificação de um único evento de falha com um atraso finito.

2.7 Diagnosticador

Em alguns casos, a ocorrência da falha pode ser detectada (observada) por sensores, por se tratarem de eventos observáveis. No entanto, na maior parte dos casos, os sensores são incapazes de detectar um evento de falha, sendo então necessário modelá-la como um evento não observável em um autômato parcialmente observado, por exemplo. Através da teoria de diagnose de falhas em SEDs, podemos verificar se a ocorrência de uma falha (evento não-observável) em um SED pode ser detectada ou não a partir da construção de um dispositivo denominado diagnosticador (SAMPATH et al., 1995). O diagnosticador é um tipo de observador, porém neste são adicionados indicadores nos estados de G, que formam os estados G_{obs}, formando assim os estados de G_d. Tais indicadores são destinados a fornecer informações sobre a possibilidade da ocorrência ou não da falha. Dois tipos de indicadores (ou marcas) são utilizados: *N*, indicando que o "evento σ_f ainda não ocorreu"e *Y* para indicar que o "evento σ_f já ocorreu". Assim, a notação dos estados $x \in X$ será do tipo *xN* ou *xY* (SAMPATH et al., 1995).

No trabalho de Sampath et al. (1995), é proposta uma metodologia para a construção deste diagnosticador. No entanto, no trabalho apresentado por Cassandras e Lafortune (2008, p. 112) é proposta uma nova metodologia, que utiliza um autômato rotulador. Esta metodologia será detalhada a seguir na Seção 2.8.

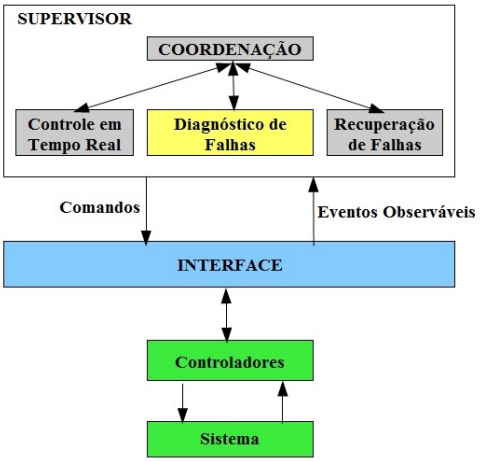
Dependendo das características do sistema e da forma como ele é modelado, as informações sobre a evolução do sistema podem estar disponíveis num único sistema de aquisição ou podem estar distribuídas entre vários sistemas (CARVALHO, 2011). Para cobrir estas duas possibilidades de configuração, foram propostas duas estruturas de diagnóstico de falhas em SEDs:

1. Diagnosticador Centralizado (SAMPATH et al., 1995), o qual usa um único diagnosticador que tem acesso a todos eventos observáveis do sistema;

2. Diagnosticador Descentralizado ou Codiagnosticador (DEBOUK et al., 2000), no qual a leitura dos sensores não está centralizada, mas distribuída entre diferentes módulos que compõem o sistema, onde cada módulo observa o comportamento do sistema usando um subconjunto do conjunto dos eventos observáveis do sistema.

No trabalho de Sampath et al. (1996) é proposta uma arquitetura genérica de um sistema que contém um subsistema para diagnóstico de falha, tal como ilustrado na Figura 2.6. Na camada inferior, tem-se o sistema e seu respectivo conjunto de controladores. Na camada superior tem-se o supervisor, que executa além das tarefas de controle o diagnóstico de falhas, a recuperação e reconfiguração do sistema após a identificação da falha e também a coordenação da operação de todos estes subsistemas. A camada de interface entre estes dois níveis é responsável por passar as informações sobre a ocorrência dos eventos observáveis do sistema ao supervisor e comunicar os comandos gerados pelo supervisor para o sistema.

Figura 2.6: Arquitetura Conceitual de um Sistema com um Subsistema de Diagnóstico de Falhas



Fonte: Adaptado de Sampath et al. (1996, p. 106)

2.8 Diagnose Centralizada

O diagnosticador G_d, é um autômato cuja definição é apresentada na equação 2.22, é obtido, conforme mencionado anteriormente, modificando-se o observador G_{obs}, com o objetivo de indicar, se possível, a ocorrência ou não de um dado evento de falha, através de marcações Y e N respectivamente.

$$G_d = (X_d, \Sigma_o, f_d, \Gamma_d, x_{0d}) \tag{2.22}$$

onde:

- X_d denota o espaço de estados do diagnosticador;
- Σ_o é o conjunto de eventos observáveis;

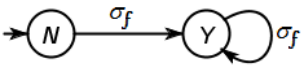
- $f_d : X_d \times \Sigma_o \rightarrow X_d$ é a função de transição de estados;
- $\Gamma_d : X \rightarrow 2^{\Sigma}$ é a função dos eventos ativos;
- x_{0d} é o estado inicial;

Pode-se construir o autômato G_d seguindo os dois passos abaixo (CASSANDRAS e LAFORTUNE, 2008, p. 112):

i) obter a composição paralela do autômato G com o autômato A_l, *i.e.*, $(G \parallel A_l)$, onde A_l é o autômato rotulador mostrado na Figura 2.7.

ii) calcular $Obs(G \parallel A_l)$, seguindo o procedimento descrito no Algoritmo 1 na Seção 2.5.

Figura 2.7: Autômato Rotulador para Construção de Diagnosticadores



Fonte: Adaptado de Cassandras e Lafortune (2008, p. 112)

Em geral, tem-se que:

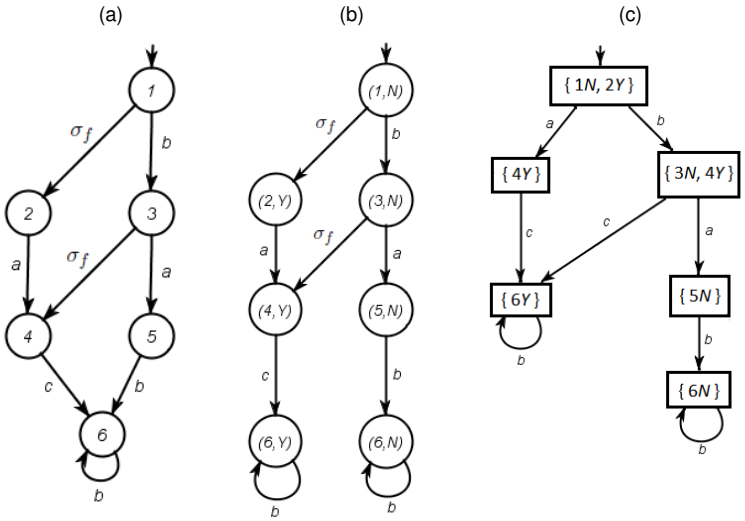
$$G_d = Obs(G \parallel A_l) \tag{2.23}$$

Note que o autômato resultante obtido no passo (i) gera a mesma linguagem de *G*, porém nesta estarão anexados rótulos nos estados de *G*, formando estados do tipo (x,Y) ou (x,N) dependendo da presença ou ausência, respectivamente, de σ_f na sequência que leva x_0 a x. Por simplicidade, geralmente (x,N) e (x,Y) são representadas como xN e xY respectivamente (CASSANDRAS e LAFORTUNE, 2008; BASILIO et al., 2010).

A seguir apresenta-se um exemplo a fim de ilustrar a construção de diagnosticadores

Exemplo 6. Considere o autômato mostrado na Figura 2.8a, onde σ_f representa o evento de falha não observável e a, b e c são os eventos observáveis do sistema. As Figuras 2.8b e 2.8c mostram a composição paralela $(G \parallel A_l)$ e o diagnosticador $G_d = Obs(G \parallel A_l)$, respectivamente.

Figura 2.8: (a) Autômato G para o exemplo 6; (b) $G \parallel A_l$; (c) $G_d = Obs(G \parallel A_l)$



Fonte: Autor

Na Figura 2.8b podemos notar que o estado 6 de G se divide em dois estados $\{6,Y\}$ e $\{6,N\}$ devido à existência de três sequências distintas $s_1 = \sigma_f ac$, $s_2 = b\sigma_f c$ e $s_3 = bab$ que leva $x_0 = 1$ para $x = 6$, onde as sequências s_1 e s_2 possuem o evento de falha σ_f. No diagnosticador (Figura 2.8c), podemos ver três tipos de estados em relação aos rótulos

Y e *N*. Os estados onde somente o rótulo *Y* está presente ($\{4Y\}$ e $\{6Y\}$) indicam um estado de falha, no qual o diagnosticador está certo sobre a ocorrência da falha. De modo similar, estados marcados apenas com rótulos do tipo *N* representam estados de não falha, para os quais o diagnosticador está certo sobre a não ocorrência da falha (estados $\{5N\}$ e $\{6N\}$). Já os estados onde ambos os rótulos *Y* e *N* estão presentes, representam estados onde o diagnosticador não pode determinar com certeza se a falha ocorreu ou não, esses estados são denominados como estados incertos (sobre a ocorrência da falha), representados neste diagnosticador pelo estados $\{1N, 2Y\}$ e $\{3N, 4Y\}$.

É importante ressaltar que, uma vez estando o diagnosticador certo sobre a ocorrência da falha, ele não retornará, *i.e.*, todos os estados seguintes continuarão indicando a falha. Entretanto, é possível que o diagnosticador mude de um estado de não falha (normal) para um estado incerto e desse para um estado certo de falha ou até mesmo normal. Formalmente os estados do diagnosticador em relação a presença dos rótulos *Y* e *N* são definidos como segue (SAMPATH et al., 1995):

Definição 16. Um estado $x_d \in X_d$ é dito certo (de falha) se $\ell = Y$ para todo $x\ell \in x_d$, e normal (ou de não falha) se $\ell = N$ para todo $x\ell \in x_d$. Se existe $(x\ell, y\tilde{\ell}) \in x_d$, *x* não necessariamente distinto de *y*, tal que $\ell = Y$ e $\tilde{\ell} = N$ então x_d é um estado incerto de G_d.

Na Figura 2.8c o estado inicial de G_d ($\{1N, 2Y\}$) é um estado incerto, porque o evento não observável σ_f pode ter ocorrido sem ter sido registrado pelo diagnosticador. Esta possibilidade é representada pela componente *2Y*, que significa que o sistema pode estar no estado 2, após a ocorrência da falha σ_f. No entanto, no estado inicial também pode ocorrer o evento observável b, de modo que o diagnosticador deve indicar que o sistema se manteve no estado 1 (devido ao evento b ainda não ter sido registado) e, portanto, a falha ainda não ocorreu, a componente *1N* indica essa possibilidade. A partir desse estado, quando o sistema reporta para o diagnosticador a ocorrência do evento *a*, ele muda para o estado $\{4Y\}$, o que torna o diagnosticador certo da ocorrência de falha

(σ_f). Uma vez neste estado, apenas o evento c será considerado pelo diagnosticador e mesmo após a sua ocorrência o diagnosticador ainda vai permanecer em um estado de falha, mas agora representado pelo estado $\{6Y\}$. Contudo, a partir do estado inicial, se o sistema informar a ocorrência do evento *b*, o diagnosticador mudará para o estado $\{3N, 4Y\}$ e ainda estará incerto sobre a ocorrência da falha. Porém, se o próximo evento que ocorrer for o evento c, então o diagnosticador passará para o estado $\{6Y\}$ tornando-se agora certo da falha, caso contrário, se o evento a ocorrer, o diagnosticador mudará para o estado $\{5N\}$ indicando que está certo da não ocorrência da falha, em outras palavras, o sistema está operando em condições normais.

A partir das definições 15 e 16, o seguinte lema é uma consequência direta entre os estados do diagnosticador e as sequências da linguagem gerada por G:

Lema 1. (SAMPATH et al., 1995) :

i) Seja $x_d = f_d(x_{0d}, s)$. Se x_d é um estado certo, então $[\forall \omega \in P_{oL}^{-1}(s)], \Sigma_f \in \omega$

ii) Se x_d é um estado incerto, então $\exists s_1, s_2 \in L$, tal que $\Sigma_f \in s_1$ e $\Sigma_f \notin s_2$, contudo $P_o(s_1) = P_o(s_2)$ e $f_d(x_{0d}, P_o(s_1)) = x_d$ e $f(x_0, s_1) \neq f(x_0, s_2)$

Da definição de estado certo (Definição 16) e do Lema 1, percebe-se que, quando o estado atual do diagnosticador é um estado certo, podemos afirmar que a falha ocorreu. No entanto, quando o diagnosticador está em um estado incerto, isso indica que há duas sequências s_1 e $s_2 \in L$ que reproduzem o mesmo registro de eventos observáveis, onde s_1 tem o evento de falha, mas s_2 não. Portanto, neste caso, baseado nos eventos observáveis não podemos concluir se a falha ocorreu.

A partir do lema anterior e da Definição 15, podemos concluir que a linguagem $L(G)$ será diagnosticável com relação a sua projeção P_o e Σ_f se, e somente se o diagnosticador sempre atingir um estado certo para todas as sequências de L que contém o evento de falha σ_f. Isso não

ocorrerá se, e somente se existir uma sequência em L que faz com que o diagnosticador fique preso em um ciclo indeterminado formado apenas por estados incertos (BASILIO et al., 2010).

As definições 17 e 18 definem formalmente estes ciclos e as condições necessárias para um conjunto de estados incertos formarem um ciclo indeterminado.

Definição 17. Seja $L(G,x) = \{s \in \Sigma^* : f(x,s) \in X\}$, ou seja, o conjunto de todas as sequências que levam o autômato G do estado $x \in X$ para um outo estado. Então um conjunto de estados $\{x_1, x_2, \ldots, x_n\} \subset X$ forma um ciclo em um autômato G se $\exists s$ tal que $s = \sigma_1\sigma_2\ldots\sigma_n \in L(G,x_1)$ onde $f(x_l, \sigma_l) = x_{l+1}, l = 1, \ldots, n-1$ e $f(x_n, \sigma_n) = x_1$ (SAMPATH et al., 1995).

Definição 18. Um conjunto de estados incertos $\{x_{d_1}, x_{d_2}, \ldots, x_{d_p}\} \subset X_d$ forma um ciclo indeterminado se:

1) $x_{d_1}, x_{d_2}, \ldots, x_{d_p}$ forma um ciclo em G_d, ou seja, existem $\sigma_l \in \Sigma_o$, $l = 1, 2, \ldots, p$, tais que $f_d(x_{d_l}, \sigma_l) = x_{d_{l+1}}$, $l = 1, 2, \ldots, p-1$ e $f_d(x_{d_p}, \sigma_p) = x_{d_1}$;

2) $\exists (x_l^{k_l}, Y), (\tilde{x}_l^{r_l}, N) \in x_{dl}$, para $x_l^{k_l}$ não necessariamente distinto de $\tilde{x}_l^{r_l}, l = 1, 2, \ldots, p, k_l = 1, 2, \ldots, m_l$, e $r_l = 1, 2, \ldots, \tilde{m}_l$ de tal forma que as sequências de estados $\{x_l^{k_l}\}$, $l = 1, 2, \ldots, p$, $k_l = 1, 2, \ldots, m_l$ e $\{\tilde{x}_l^{r_l}\}$, $l = 1, 2, \ldots, p$, $r_l = 1, 2, \ldots, \tilde{m}_l$, podem ser dispostas de modo a formar ciclos em G, cujas sequências correspondentes s e $\tilde{s}$, formadas com eventos que definem a evolução dos ciclos, têm como projeção $\sigma_1\sigma_2, \ldots, \sigma_p$, onde $\sigma_1, \sigma_2, \ldots, \sigma_p$ são definidos de acordo com o item 1 (BASILIO et al., 2010).

Através das definições 15, 17 e 18 e do Lema 1, pode-se apresentar a seguinte condição necessária e suficiente para o diagnóstico de uma linguagem (BASILIO et al., 2010; SAMPATH et al., 1995).

Teorema 1. Uma linguagem *L* gerada por um autômato *G* será diagnosticável em relação a sua projeção P_o e $\Sigma_f = \{\sigma_f\}$ se, e somente se, seu diagnosticador G_d não possuir ciclos indeterminados.

Para o autômato do Exemplo 6, note que o diagnosticador G_d, mostrado na Figura 2.8c não possui ciclos indeterminados nem estados ambíguos, então pode-se concluir que *L* é diagnosticável em relação a P_o e $\Sigma_f = \{\sigma_f\}$.

Exemplo 7. Considere agora o autômato *G* da Figura 2.9a. Considere também que $\Sigma = \{a, b, c, \sigma_f\}$, onde $\Sigma_o = \{a, c\}$ o conjunto de eventos observáveis, $\Sigma_{uo} = \{b, \sigma_f\}$ o conjunto de eventos não observáveis e $\Sigma_f = \{\sigma_f\}$, onde σ_f representa o evento de falha.

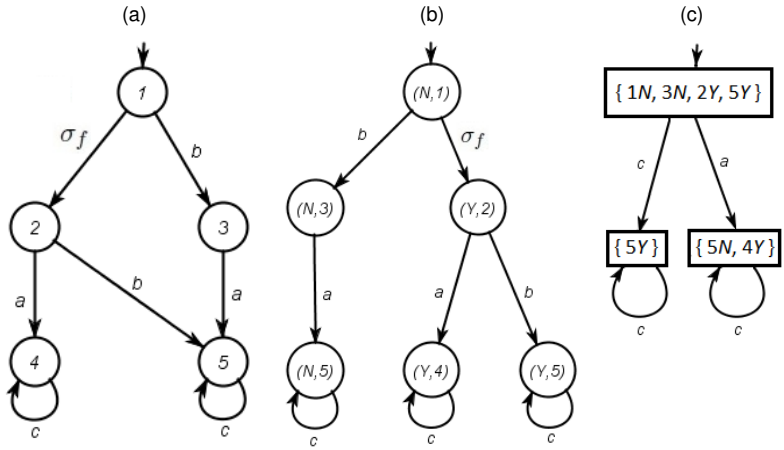
A Figura 2.9b mostra a composição paralela de $G \parallel A_l$ e o respectivo diagnosticador G_d de *G* é mostrado na Figura 2.9c. Note que o estado $\{5N, 4Y\}$ é um estado incerto e como $f_d(\{5N, 4Y\}, c) = \{5N, 4Y\}$, então o estado incerto $\{5N, 4Y\}$ forma um ciclo indeterminado em G_d. Este ciclo é devido aos dois ciclos em *G* formado pelos estados 4 e 5, como se pode ver na Figura 2.9a. Pode-se concluir então, que neste caso *L* é não diagnosticável em relação a P_o e Σ_f

Observação A presença de ciclos em G_d formada apenas por estados incertos não significa, necessariamente a impossibilidade de diagnosticar a ocorrência de falha. Para que *L* seja não diagnosticável em relação a P_o e Σ_f, é necessário que após a ocorrência de falha, exista em *G* um ciclo de estados que corresponda ao ciclo de estados incertos em G_d (SAMPATH et al., 2005; BASILIO et al., 2010). Em Cassandras e Lafortune (2008, p. 114) é mostrado um exemplo que ilustra esta condição.

2.9 Diagnose Descentralizada com Coordenação

O diagnóstico centralizado não é aplicável a todos os tipos de sistemas, principalmente àqueles que possuem uma característica distri-

Figura 2.9: (a) Autômato G para o exemplo 7; (b) $G \,||\, A_l$; (c) Diagnosticador Centralizado de G



Fonte: Autor

buída, uma vez que o diagnosticador não terá acesso a todos os conjuntos de eventos necessários para diagnose (BASILIO e LAFORTUNE, 2009).

Considerando este cenário, foi proposto por Debouk et al., (2000) uma estrutura descentralizada com coordenação, chamada codiagnose. Esta abordagem utiliza módulos locais onde cada módulo é capaz de observar parte dos eventos observáveis do sistema. Esses módulos locais se comunicam com um coordenador, o qual é responsável pelo diagnóstico de falhas. Mais tarde, no trabalho proposto por Contant et al., (2006) o conceito de diagnosticabilidade modular foi introduzido pela primeira vez para um sistema que pode ser modelado pela composição paralela de autômatos, onde cada autômato representa um componente local (ou subsistema) do sistema global. Assim, se cada subsistema é diagnosticável, então o diagnóstico de falhas para o sistema global será obtido

utilizando apenas os diagnosticadores locais (BASILIO et al., 2010). Em Basilio e Lafortune (2009), foi formalmente apresentado o conceito de codiagnose robusta, suas propriedades e as condições necessárias e suficientes para codiagnose robusta; onde um sistema descentralizado com codiagnose será dito robusto se e somente se após a perda de comunicação entre um ou mais módulos e o coordenador, o sistema ainda continuar sendo diagnosticável (BASILIO et al., 2010).

Conforme mostrado na Figura 2.10 (BASILIO e LAFORTUNE, 2009), as leituras dos sensores não são centralizadas, mas distribuídas entre diferentes módulos M_i, $i = 1,2,\ldots,N$. Cada módulo possui seu próprio conjunto de eventos observáveis Σ_{oi}, $i = 1,2,\ldots,N$, conforme os sensores a ele conectados. Assim, cada módulo observa o comportamento do sistema com base nas informações fornecidas pelos sensores a ele conectados e processa o diagnóstico, em seguida, de acordo com a estrutura descentralizada proposta por Debouk et al., (2000), cada módulo comunica seu diagnóstico somente ao coordenador. O coordenador, por sua vez processa a informação conforme um conjunto de regras pré-estabelecidas e toma a decisão sobre a ocorrência ou não de falha (BASILIO et al., 2010).

É importante notar que o coordenador é, em geral um sistema dedicado e sem conhecimento do modelo do sistema, possuindo então capacidades de memória e de processamento limitados (BASILIO e LAFORTUNE, 2009).

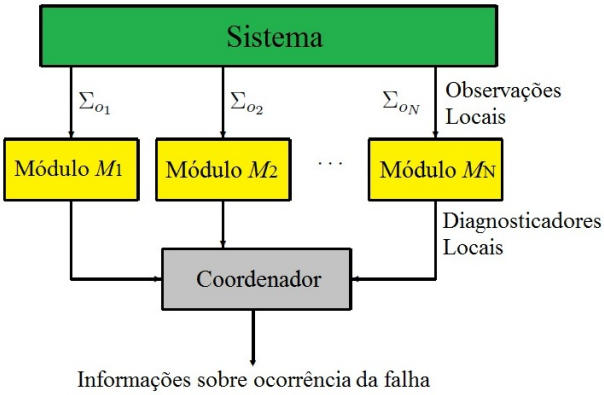
O diagnóstico da linguagem L pela estrutura descentralizada com coordenação, mostrada da Figura 2.10, depende, além do conjunto de falha Σ_f, de outros quatro elementos (BASILIO e LAFORTUNE, 2009):

i) do conjunto de regras usado para gerar os diagnósticos locais;

ii) das regras de comunicação entre módulos e coordenador;

iii) das regras de decisão de diagnóstico empregadas pelo coordenador;

Figura 2.10: Arquitetura Descentralizada com Coordenação



Fonte: Adaptado de Basilio e Lafortune (2009, p. 2205)

iv) das projeções $P_{o,i} : \Sigma \to \Sigma_{o,i}$, $i = 1,2,\ldots N$ associadas a cada módulo M_i.

O conjunto de elementos (i), (ii) e (iii) são geralmente referidos na literatura como *protocolo* (BASILIO e LAFORTUNE, 2009).

A Definição 15 pode ser modificada para incluir coordenação com sistemas descentralizados, como segue (DEBOUK et al., 2000).

Definição 19. Uma linguagem L viva e prefixo-fechada é dita diagnosticável em relação a um protocolo, um conjunto de projeções $P_{o,i}, i = 1,\ldots,n$, e a um conjunto de falhas $\Sigma_f = \{\sigma_f\}$, se a seguinte condição for verdadeira:

$$(\exists n \in \mathbb{N})(\forall s \in \Psi(\Sigma_f))(\forall t \in L/s)(\| t \| \geq n \Rightarrow C \textit{ está certa}),$$

em que *C* denota a informação passada para o coordenador para fazer

o diagnóstico e para cada caminho do sistema, C é representado por um conjunto de informações que é dependente do protocolo usado.

A condição de diagnosticabilidade definida acima, requer que a detecção de qualquer falha seja efetuada pelo coordenador após um atraso finito de sua ocorrência (DEBOUK et al., 2000).

Três protocolos foram propostos por Debouk et al. (2000), em particular, para o protocolo 3, um diagnosticador parcial é implementado em cada módulo, cujo estado representa a informação diagnóstico após a ocorrência de um evento observável. Quando um módulo observa um evento que leva o seu diagnosticador para um estado certo, então este comunica a ocorrência da falha somente para o coordenador. O coordenador entretanto, declara a ocorrência de uma falha, sempre que pelo menos um dos módulos comunicar uma ocorrência de falha, caso contrário, se não há relato de ocorrência de falhas de qualquer módulo, ele permanece em silêncio. O protocolo 3 pode ser considerado como uma extensão do diagnóstico centralizado para diagnóstico descentralizado com coordenação, e, portanto, ele também é conhecido como *codiagnose* e os diagnosticadores locais chamados de diagnosticadores parciais (BASILIO e LAFORTUNE, 2009).

2.10 Conclusões do Capítulo

Neste capítulo foram apresentados os conceitos básicos e principais resultados acerca do estudo da diagnose de falhas em SEDs. Foram apresentadas as condições necessárias e suficientes para diagnose, tanto para o caso centralizado como para o caso descentralizado com coordenação (codiagnose). Também foi mostrada uma metodologia para construção de diagnosticador através de um autômato rotulador. Nos capítulos seguintes estes conceitos serão aplicados para a diagnose de falhas em sistemas embarcados, e uma metodologia de diagnóstico multicamadas é proposta e, posteriormente, aplicada a um estudo de caso.

3 PROPOSTA DE ARQUITETURA MULTICAMADAS PARA O DIAGNÓSTICO DE FALHAS EM SISTEMAS EMBARCADOS

A complexidade dos dispositivos eletrônicos que fazem parte do nosso cotidiano está em constante crescimento e é particularmente emergente nos sistemas embarcados, onde um novo recurso ou uma nova variante do equipamento pode ser rapidamente introduzida no mercado apenas atualizando o *software* atual. Este tipo de abordagem está cada vez mais presente em produtos eletrônicos tais como telefones celulares, eletrodomésticos, automóveis e outros. Em combinação com a redução de *time-to-market* (tempo para lançamento) o tempo de projeto também tem diminuído sistematicamente, o que leva à necessidade de introduzir técnicas de diagnóstico de falhas mais eficientes a fim de identificar e corrigir as falhas que podem ocorrer tanto durante a fase de concepção quanto após o lançamento e, assim, garantir não só que o produto seja lançado no prazo desejado, como também com a qualidade exigida (ZOETEWEIJ et al., 2008). No que diz respeito ao pós-lançamento, ou seja, no período em que o produto está em uso pelo consumidor, um melhor diagnóstico ajuda não só a identificar e entender rapidamente um modo de falha e, assim prontamente corrigi-lo, mas também usar as informações obtidas no diagnóstico de falhas para mudar os próximos projetos a fim de evitar a ocorrência das mesmas. Quando se tratam de produtos eletrônicos de consumo, após o lançamento, uma falha não identificada anteriormente pode ser muito prejudicial para a empresa, tanto em termos financeiros devido ao custo com cobertura de garantias ou até mesmo um *recall*, como em termos de imagem da marca associada a este equipamento ou produto. Isso torna o diagnóstico de falhas algo ainda mais relevante e tem levado as grandes empresas a investir cada vez mais dentro dos projetos em soluções e ferramentas para diagnóstico e recuperação de falhas.

Frente a esse problema, tanto o projeto de diagnosticadores bem como a definição de uma estrutura de sistema de diagnóstico se torna algo crucial e complexo, o que requer o uso de técnicas e sistemáticas apuradas e adequadas para resolver este problema. A busca por tais técnicas e sistemáticas tornaram-se os principais motivadores deste trabalho. Neste capítulo será inicialmente apresentado uma breve introdução sobre sistemas embarcados e em seguida será apresentada uma proposta de arquitetura multicamadas para diagnóstico voltada a sistemas embarcados.

3.1 Sistemas Embarcados

Um sistema embarcado é um sistema microprocessado no qual o computador é completamente dedicado ao dispositivo ou sistema que ele controla (EMBARCADOS, 2014). Diferentemente dos computadores de propósito geral, como o computador pessoal, um sistema embarcado realiza um conjunto de tarefas predefinidas e com requisitos específicos. Em geral, tais sistemas não podem ter sua funcionalidade alterada durante o uso e o usuário final não tem acesso ao programa que foi embutido no dispositivo. Em alguns casos, o sistema também é executado com recursos computacionais limitados: sem teclado, sem tela e com pouca memória e possuem restrições para computação em tempo real por questões de segurança ou usabilidade. O *software* escrito para sistemas embarcados é muitas vezes chamado *firmware*, e é armazenado em uma memória ROM ou memória *Flash* ao invés de um disco rígido.

O primeiro sistema embarcado reconhecido foi o *Apollo Guidance Computer*, desenvolvido por Charles Stark Draper no MIT e utilizado no projeto Apollo (HALL, 1996). Já o primeiro sistema embarcado de produção em massa foi o computador guia do míssil nuclear LGM-30 Míssil *Minuteman*, lançado em 1961, que possuía um disco rígido como memória principal. Em sua segunda versão em 1966, o computador guia foi substituído por um novo, que constituiu o primeiro uso em grande volume de circuitos integrados. Desde suas primeiras aplicações na década

de 1960, os sistemas embarcados vêm reduzindo de preço e crescendo em poder de processamento e funcionalidade. Em meados da década de 1980, vários componentes externos foram integrados no mesmo *chip* do processador, dando origem ao *single-chip*, o que resultou em circuitos integrados chamados microcontroladores e na difusão dos sistemas embarcados. Com o baixíssimo custo dos microcontroladores, tornou-se viável substituir componentes analógicos caros como potenciômetros e capacitores por eletrônica digital controlada por pequenos microcontroladores. No final da década de 1980, os sistemas embarcados já eram a regra ao invés da exceção em dispositivos eletrônicos. A indústria automobilística começou a fazer uso do microprocessador tão logo as CPUs em *single-chip* tornaram-se disponíveis. Segundo Wolf (2008), até meados de 2008 o uso mais importante e sofisticado de microprocessadores em automóveis foi para controlar o motor determinando, quando as velas de ignição deveriam gerar as faíscas e controlando-se a mistura de combustível e ar, para aumentar a eficiência do motor. Dada a grande variedade de tipos de microcontroladores disponíveis hoje, não é surpresa que eles estejam presentes em quase tudo que nos cerca e fazemos uso em nosso cotidiano. Além dos automóveis, podemos citar os eletrodomésticos, tais como forno de micro-ondas, geladeiras, fogões e televisões. Estas, por sua vez, fazem um uso extensivo de processadores embarcados e, em muitos casos, CPUs especializadas são especialmente projetadas para executar algoritmos críticos, como por exemplo a CPU projetada para o processamento de áudio no *chip set* SGS Thomson para *DirecTV* (LIEM et al., 1998). Podemos citar ainda alguns dispositivos de uso pessoal, como celulares, *Smart Phones* e *I-Pods*, etc.

Quanto à sua aplicação, um sistema embarcado pode ser classificado como:

• **De Propósito geral**: são as aplicações mais parecidas com os *desktops*, porém estão montados em "embalagens"específicas. Caracterizam-se pela grande interação entre o usuário e o sistema, geralmente através de terminais de vídeo ou monitores. Como exemplo têm-se os videogames, os conversores de TV a cabo e caixas de bancos;

• **Sistemas de controle**: realizam controles em malha fechada com realimentação em tempo real. Geralmente são as aplicações mais robustas, com placas dedicadas e múltiplos sensores de entrada e saída. Muitas vezes fornecem pouca interação com o usuário, mostrando sinalizações através de LEDs ou *displays*. Usados para o controle de motores de automóveis, processos químicos, controle de voo, usinas nucleares, etc.;
• **Processamento de sinais**: onde envolve um grande volume de informação a ser processada em curto espaço de tempo. Os sinais a serem tratados são digitalizados por meio de conversores Analógicos-Digitais, processados, e novamente convertidos em sinais analógicos. Caso de tratamento de áudio, filtros, modems, compressão de vídeo, radares, sonares, etc.;
• **Comunicações e redes**: em aplicações que evolvem chaveamento e distribuição de informações, tais como sistemas de telefonia, telecomunicações e internet.

Dois modos de funcionamento dos sistemas embarcados são determinantes para saber como projetar o dispositivo e como será seu funcionamento e comportamento na aplicação para o qual foi desenhado:
• **Reativo**: o funcionamento se dá como resposta a eventos externos, que podem ser periódicos (de controles de *loop*) ou assíncronos (pressionamento de um botão por parte do usuário, por exemplo). Há, portanto, uma necessidade de entrada de dados para que aconteçam as ações de funcionamento;
• **Controle em tempo real**: existem limites de tempo para executar cada tarefa (leitura de sensor, emissão de sinais para um atuador, atualização de *display*, etc.). Este modo de operação, por ser cíclico, não depende da entrada de sinais para executar as atividades, sendo inclusive capaz de tomar decisões referentes à ausência dos mesmos.

Os sistemas de tempo real são classificados em:

i) ***Soft Real Time***: As tarefas podem ser executadas em um intervalo de

tempo específico, sem consequências graves se este limite de tempo não for cumprido. Um exemplo é um sistema bancário, onde apenas uma mensagem de erro aparecerá se determinada tarefa não for realizada dentro do tempo pré-determinado.

ii) ***Hard Real Time***: As tarefas devem ser executadas em um tempo específico, com consequências graves se qualquer tarefa falhar. Como exemplo pode-se pensar nos sistemas de controle de um avião, onde uma falha pode resultar em queda da aeronave e perdas de vidas.

3.2 Estrutura de Sistemas Embarcados

Analisando de maneira sistêmica, pode-se perceber que os diferentes equipamentos ou dispositivos que são controlados por um sistema embarcado são em sua grande maioria compostos pelos seguintes elementos:

• ***Hardware***: placa eletrônica que contém um ou mais microcontroladores, também circuitos de condicionamento de sinais além dos circuitos de potência e de acionamento de cargas e atuadores;
• ***Software***: conjunto de algoritmos para executar rotinas tais como leitura de sensores, tratamento matemático dos sinais, lógica de controle, rotinas de aplicação e interação com o usuário, dentre outras;
• **Cargas e Sensores**: cargas e atuadores propriamente ditos tais como motores, válvulas e *displays*. Dentre os sensores podem-se citar os sensores de temperatura, pressão, infravermelho, calor ou ainda simples chaves como chaves de fim de curso e também teclados.

Para exemplificar o contexto acima, tomemos dois equipamentos bem distintos uma lavadora de roupas e um aparelho de TV. A seguir, para ambos os equipamentos, são mostrados alguns dos componentes presentes de cada um dos elementos *Hardware*, *Software*, Cargas e Sensores:

Hardware:
- *TV*: Filtros analógicos, circuitos amplificadores, condicionadores de sinais de áudio e vídeo, circuitos de potência para auto-falantes e para tela (*display* de LCD, LED, Plasma, etc.);
- *Lavadora*: Circuitos de potência para acionamento do motor, condicionadores de sinais dos sensores, circuitos de acionamento para válvulas e bombas.

Software:
- *TV*: Processamento de imagem e som, menus do usuário, controle da matriz da tela (*display* de LCD, LED, Plasma, etc.);
- *Lavadora*: Rotinas de controle de motor, processamento dos sinais dos sensores, rotinas de programas de lavagem.

Cargas e sensores:
- *TV*: Tela (*display* de LCD, LED, Plasma, etc.), auto-falantes, sensor de infravermelho do controle remoto;
- *Lavadora*: Motor, válvulas, bombas, sensores de nível e pressão.

Na Figura 3.1a apresenta-se o diagrama de blocos para um dispositivo com sistema de controle embarcado.

Pode-se organizar estes elementos (*Hardware*, *Software*, Cargas e Sensores) em três categorias distintas que chamaremos de *Camada de Hardware*, *Camada de Software* e *Camada de Cargas e Sensores*. Desta forma, neste trabalho representa-se a estrutura básica de um sistema embarcado em camadas, conforme ilustrado na Figura 3.1.

De maneira geral temos:

• **Camada de *Software*** : onde são implementados os algoritmos de controle, tratamento de sinais, aplicação, etc.;
• **Camada de *Hardware***: constituída pelos componentes e circuitos que

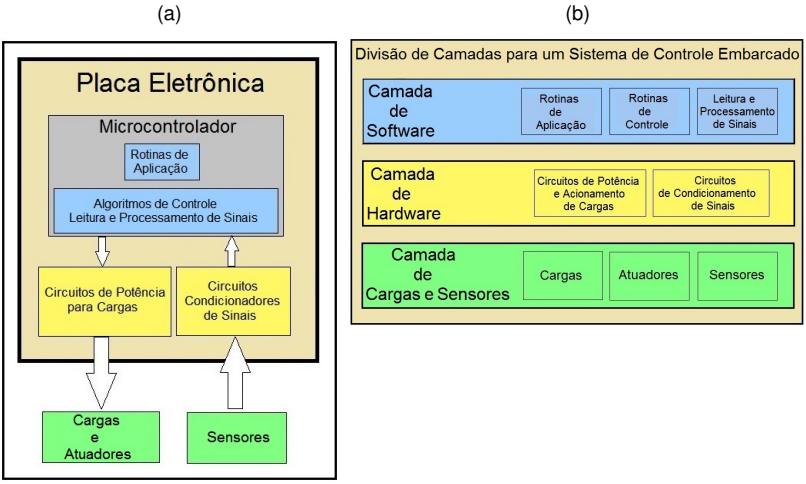
compõem a placa eletrônica, tais como relés, circuitos de potência, circuitos de condicionamento de sinais, microcontroladores, etc.;
• **Camada de Cargas e Sensores**: composta pelas cargas ou atuadores, tais como motores, válvulas, etc., e também pelo sensores.

Esta divisão é adequada tanto sob o ponto de vista do equipamento quanto do ponto de vista do projeto. Nas grandes empresas o projeto de sistemas embarcados é dividido em equipes com qualificações específicas para trabalhar em cada uma destas camadas, ou seja, uma equipe mais especializada no desenvolvimento do *software*, outra mais especializada no projeto do *hardware* e outra com conhecimentos específicos nas características elétricas e funcionais das cargas, atuadores e sensores que serão aplicadas ao projeto. Esta divisão permite ainda estabelecer uma metodologia de desenvolvimento de projeto que seja flexível e maximize o reuso dos circuitos, rotinas de software, cargas, etc. desenvolvidos no projeto. Desta forma é possível por exemplo que a equipe de desenvolvimento de *software* construa toda uma arquitetura onde seja possível atender diferentes tipos de *hardware* e cargas com um mínimo ou sem nenhuma parametrização, ou ainda, que a equipe de desenvolvimento de *hardware* possa projetar circuitos que atenderam diferentes especificações de cargas e sensores. Estabelecendo-se padrões, interfaces definidas e especificações claras é possível ainda que o projeto de cada uma das camadas ocorra de maneira independente e com pouca necessidade de interação entre as equipes, prática muito comum em empresas que possuem equipes de trabalho descentralizadas, muitas vezes até localizadas em países diferentes.

3.3 Diagnóstico Multicamadas

Normalmente, nos sistemas de diagnóstico tradicionais, devido ao efeito ou característica da falha, nem sempre é possível identificar a real origem desta falha. Para ilustrar isso, tomemos como exemplo novamente uma lavadora de roupas, onde temos uma bomba usada para

Figura 3.1: (a) Diagrama de blocos para um dispositivo com sistema de controle embarcado; (b) Divisão de camadas para um dispositivo com sistema embarcado



Fonte: Autor

permitir enchimento de água. No caso de uma falha onde a lavadora não consegue encher de água o cesto, para assim iniciar um ciclo de lavagem, temos como potenciais fontes de falha:

i) a própria bomba, isto é, uma possível falha na carga;

ii) sensor de vazão danificado, ou seja, uma falha de sensor;

iii) um problema elétrico no placa eletrônica, tal como um componente ou circuito danificado, não permitindo assim o acionamento da bomba, ou seja, neste caso temos uma falha no *hardware*;

iv) um erro ou inconsistência no algoritmo de controle da bomba, que pode ser causado tanto por um erro de lógica no *software* ou pelo

corrompimento de algum dado da memória, devido a uma interferência eletromagnética por exemplo, que cause uma falha no *software* e impeça o acionamento da bomba.

Em um sistema de diagnóstico comum, esta falha seria caracterizada simplesmente como uma falha do sistema de enchimento, não permitindo distinguir a real origem da mesma, uma vez que no próprio diagnóstico não são distinguidas as interações entre *software*, *hardware* e carga e apenas o efeito final da falha no sistema como um todo é observado, ou seja, o cesto não encheu de água.

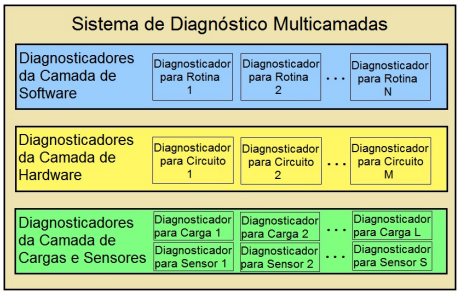
Uma vez que cada camada possui funções e características bem distintas umas das outras e, com o intuito de identificar precisamente a origem da falha, propõe-se a utilização de diagnosticadores em cada uma destas camadas para assim assegurar que a falha está por exemplo no *hardware* do circuito de acionamento da bomba, e não no *software* de controle ou ainda na própria bomba, permitindo então que uma possível ação de recuperação ou correção possa ser corretamente aplicada.

Na Figura 3.2 mostra-se de maneira genérica como estes diagnosticadores podem ser distribuídos em cada uma destas camadas, dando origem ao que denominamos neste trabalho de Sistema de Diagnóstico Multicamadas. O objetivo do diagnóstico multicamadas é permitir uma maior descriminação quanto à origem da falha, possibilitando assim um diagnóstico mais preciso e robusto.

Em outras palavras, os diagnosticadores alocados da *Camada de Software* são responsáveis por diagnosticar falhas referentes às rotinas ou algoritmos de *software*, como desvio do fluxo de *software*, mudanças de estados não permitidas ou não previstas, etc.

Já os diagnosticadores alocados na *Camada de Hardware* são responsáveis por diagnosticar falhas em circuitos ou componentes elétricos, como por exemplo falha de relés, em circuitos de potência, circuitos de acionamentos, circuitos de condicionamentos de sinais, etc.

Figura 3.2: Sistema de Diagnóstico Multicamadas



Fonte: Autor

Por fim, os diagnosticadores da *Camada de Cargas e Sensores* serão responsáveis por identificar quando uma carga ou sensor em específico falhou como, por exemplo, um motor danificado que não dá partida, uma válvula que não aciona ou ainda um sensor cujo sinal gerado está fora do esperado ou admitido.

O grande objetivo por trás do diagnóstico multicamadas é assegurar que a origem da falha seja realmente identificada com a maior discriminação possível, assim podemos dizer que o sistema de diagnóstico multicamadas aqui proposto será um sistema de diagnóstico multicamadas determinístico se, e somente se, a condição abaixo for satisfeita:

Definição 20. *(Sistema de Diagnóstico Multicamadas Determinístico):* Seja $\Sigma_f = \Sigma_{f_{sw}} \bigcup \Sigma_{f_{hw}} \bigcup \Sigma_{f_{cs}}$, o conjunto de falhas do sistema, onde $\Sigma_{f_{sw}} = \{\sigma_{sw_i}\}$, com $i \in I = 1,...,s$, $\Sigma_{f_{hw}} = \{\sigma_{hw_j}\}$, com $j \in I = 1,...,h$ e $\Sigma_{f_{cs}} = \{\sigma_{cs_k}\}$, com $k \in I = 1,...,c$, denotam respectivamente os conjuntos de falhas nas camadas de *software*, *hardware* e de cargas e sensores, e s, h e c denotam o número de falhas nestes conjuntos. Sejam ainda $G_{d_{sw_i}}$, $G_{d_{hw_j}}$ e $G_{d_{cs_k}}$ os diagnosticadores para as falhas do tipo σ_{sw_i}, σ_{hw_j} e σ_{cs_k}, respectivamente.
Um conjunto de diagnosticadores com as características descritas é dito

ser um Sistema de Diagnóstico Multicamadas Determinístico se a ocorrência de uma falha $\sigma_f \in \Sigma_f$ é identificada somente pelo seu respectivo diagnosticador ($G_{d_{sw_i}}$, $G_{d_{hw_i}}$ ou $G_{d_{cs_i}}$), permanecendo os demais inalterados.

Para exemplificar a Definição 20, vamos retomar o exemplo da lavadora. No caso de uma falha na bomba de enchimento, onde a mesma não ligue por estar com enrolamento da bobina rompido, o diagnosticador desta bomba, alocado na *Camada de Cargas e Sensores*, deve detectar esta falha, e não será permitido, por exemplo, que o diagnosticador do circuito de acionamento da bomba alocado na *Camada de Hardware* detecte esta falha, confundindo como uma falsa falha de *hardware* ou ainda que um diagnosticador da rotina de acionamento desta bomba alocado na *Camada de Software* diagnostique a mesma falha, confundindo como uma falsa falha de *software*.

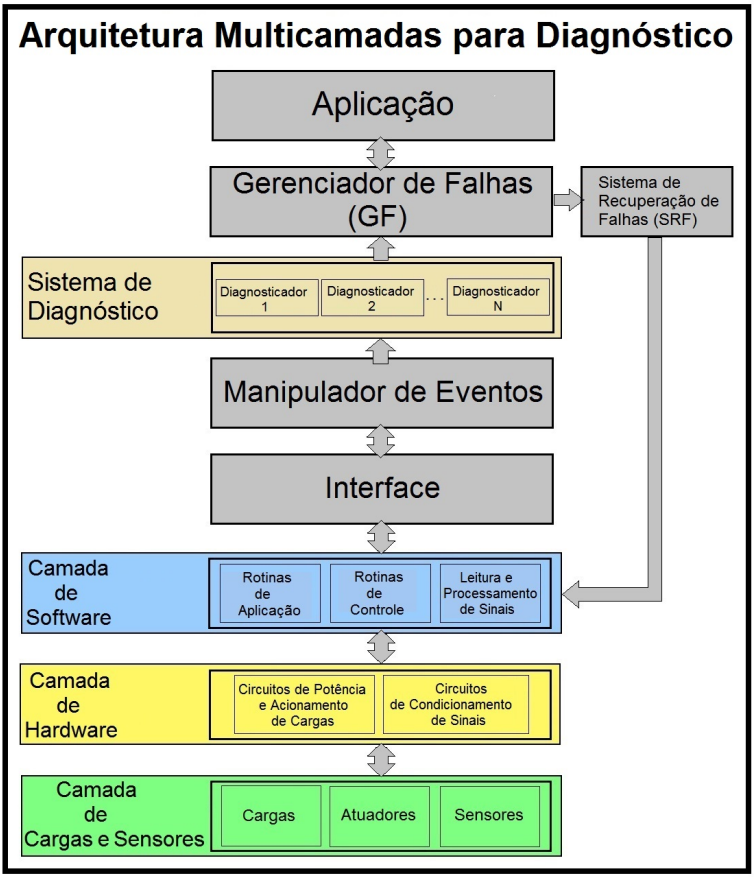
3.4 Arquitetura Multicamadas para Diagnóstico de Falhas em Sistemas Embarcados

Para que o sistema de diagnóstico mostrado na Figura 3.2 possa ser embarcado no dispositivo ou equipamento no qual se deseja realizar o diagnóstico de falhas, e ainda, para que o diagnóstico possa ser realizado, é necessário criar uma arquitetura que permita que estes diagnosticadores acessem todas as informações necessárias para tanto.

Muitas vezes, após a ocorrência de uma falha é importante que alguma ação de recuperação seja tomada pelo sistema, assim quanto mais preciso for o diagnóstico quanto à origem da falha, mais efetiva será esta ação de recuperação. Em alguns casos também se faz necessário exteriorizar de algum modo (via *display*, LEDs, etc.) a ocorrência da falha, para fins de manutenção corretiva, por exemplo. Portanto, novamente quanto mais preciso for o diagnóstico, mais eficaz será a intervenção técnica para manutenção.

Assim, no intuito de atender a estas condições, neste trabalho apresenta-se uma arquitetura multicamadas de diagnóstico de falhas em sistemas embarcados. Esta arquitetura é ilustrada por intermédio da Figura 3.3 e é detalhada a seguir.

Figura 3.3: Arquitetura Multicamadas para Diagnóstico



Fonte: Autor

Começando de baixo para cima, temos as três camadas anterior-

mente definidas: *Camada de Cargas e Sensores*; *Camada de Hardware*; *Camada de Software*. Nestas camadas, estão os elementos que compõem um sistema embarcado propriamente dito, conforme apresentado na Seção 3.2. Na sequência temos o bloco *Interface* o qual é responsável por requisitar e gerenciar as informações necessárias provenientes das rotinas de controle, dos sensores e das cargas, destes dois últimos previamente tratadas pelos circuitos de condicionamento de sinais da *Camada de Hardware* e processadas na *Camada de Software*. Estas informações são então transmitidas ao bloco *Manipulador de Eventos*. Este bloco por sua vez é responsável por "manipular"corretamente essas informações a fim de fornecer os eventos necessários para os diagnosticadores. É importante mencionar também que este bloco pode, se necessário, atuar como um "sensor virtual", combinando duas ou mais informações para criar um evento no formato esperado pelos diagnosticadores.

Logo acima, temos o bloco denominado *Sistema de Diagnóstico*, onde os diagnosticadores são implementados de forma modular. Os diagnosticadores recebem os eventos do bloco *Manipulador de Eventos*, processam e comunicam seu diagnóstico ao bloco *Gerenciador de Falhas (GF)*.

O *Gerenciador de Falhas (GF)* gerencia as informações de falhas reportadas por cada diagnosticador antes de comunicá-la à *Aplicação* e ao bloco denominado *Sistema de Recuperação de Falhas (SRF)*. Dentre os principais atributos do *GF* destacam-se: a capacidade de gerar códigos de identificação de falha, organizar as falhas em uma lista de acordo com uma prioridade pré estabelecida, gerar um *log* com uma "fotografia"das condições do sistema no momento em que uma falha foi relatada por qualquer diagnosticador, por exemplo.

O bloco *SRF* é responsável por executar as rotinas de recuperação de falhas, quando necessário. Este bloco contém as rotinas de recuperação para cada tipo de falha que pode ser reportada pelos diagnosticadores via *GF* e também possui as regras de como e quando aplicar cada uma destas rotinas.

Finalmente, no topo temos o bloco de *Aplicação*, onde podem ser implementadas as rotinas e métodos necessários para exteriorizar as informações referentes às falhas e reiniciar as condições de falhas, para o caso de uma manutenção corretiva por exemplo. Portanto este bloco é responsável por proporcionar uma interação com o usuário externo.

Como se pode notar, esta arquitetura considera a utilização de um conjunto de diagnosticadores conectados a um coordenador (bloco GF, neste caso), que se assemelha à diagnose descentralizada com coordenação proposta por Debouk et al. (2000) (ver Figura 2.10). No entanto, a arquitetura aqui proposta difere daquela, pois considera-se cada diagnosticador como um diagnosticador centralizado, como proposto por Sampath et al., (1995), já que cada diagnosticador é responsável por diagnosticar uma falha em específico e tem acesso a todos os eventos necessários do sistema para a realização do diagnóstico. Pode-se considerar como uma extensão da arquitetura mostrada na Figura 2.6 onde cada diagnosticador é uma instância adicional do bloco de Diagnóstico de Falhas

3.5 Conclusões do Capítulo

Neste capítulo foram apresentados os conceitos básicos e as principais características dos sistemas embarcados. Propôs-se que os elementos de um sistema embarcado fossem organizados em três camadas: *Camada de Software*, *Camada de Hardware* e *Camada de Cargas e Sensores*, onde cada uma destas camadas possui características, funcionalidades e finalidades distintas. Viu-se que esta divisão por camadas não só traz facilidades em termos de metodologia de projeto como também permite que diagnosticadores possam ser devidamente desenvolvidos para realizar um diagnóstico mais preciso que permita uma melhor descriminação da origem da falha. Por fim, foi apresentado uma estrutura de diagnóstico completa, onde seu principal objetivo é viabilizar a implementação do sistema de diagnóstico multicamadas e também pro-

ver meios para que após identificada uma falha, a mesma possa ser tratada por uma rotina de recuperação, se necessário, ou ainda, que seja possível exteriorizar sua ocorrência e também qualquer tipo de informação desejável referente a esta falha. É importante ressaltar que tanto o sistema de diagnóstico multicamadas como a arquitetura de diagnóstico apresentadas são independentes da metodologia a ser empregada para a construção dos diagnosticadores ou da arquitetura de *software* utilizada no sistema embarcado.

No próximo capítulo será apresentado um estudo de caso para um refrigerador *Frost Free* constituído por um sistema embarcado, onde será aplicado tanto o sistema de diagnóstico multicamadas como a arquitetura de diagnóstico apresentadas neste capítulo. Os diagnosticadores serão projetados no contexto de diagnose de falhas em SEDs conforme proposto por Sampath et al. (1995). Posteriormente, no Capítulo 5 serão apresentados os detalhes da implementação em *software* destes diagnosticadores, segundo a arquitetura apresentada na Figura 3.3.

4 ESTUDO DE CASO: REFRIGERADOR DOMÉSTICO *FROST FREE*

No capítulo anterior foi apresentado um sistema de diagnóstico multicamadas, bem como uma arquitetura de diagnóstico onde este sistema pode ser aplicado. Neste capítulo é apresentado um estudo de caso para um refrigerador doméstico do tipo *Frost Free*, onde este sistema de diagnóstico é aplicado. O refrigerador em questão é controlado por um sistema embarcado constituído de um microcontrolador juntamente com uma placa de controle onde há diversos circuitos que vão desde condicionadores de sinais até circuitos de acionamento das cargas e também possui certa diversidade em termos de sensores e tipos de cargas. Já em relação ao *software*, têm-se diferentes rotinas que vão desde o controle de acionamento de cargas até o controle da rotina termostática e de degelo. Por possuir esta diversidade de elementos em termos de *hardware*, *software* e cargas, que podem estar sujeitos a falhas, o refrigerador em questão se mostra adequado para ser utilizado como estudo de caso para validação da arquitetura multicamadas para o diagnóstico preciso de falhas, proposto no capítulo anterior, pois permite explorar as características desta arquitetura. Os diagnosticadores foram projetados no contexto de diagnose de falha em sistemas a eventos discretos, conforme proposto por Sampath et al. (1996), uma vez que o comportamento do sistema do refrigerador proposto neste estudo de caso pode ser, com certo nível de abstração, modelado como SED.

Nas seções seguintes serão apresentadas as informações relevantes sobre o funcionamento de um refrigerador *Frost Free*, necessários para o desenvolvimento deste estudo de caso, serão apresentados os elementos de cada uma das camadas (*hardware*, *software* e cargas) para qual se deseja realizar o diagnóstico de falhas e, por fim, será detalhado o projeto dos diagnosticadores de cada um desses elementos.

4.1 Descrição do Refrigerador *Frost Free*

4.1.1 Principio de Funcionamento de um Refrigerador

Um refrigerador, como o mostrado na Figura 4.1 da fabricante *Whirlpool*, de um modo geral funciona em ciclos, usando um fluído refrigerante num circuito fechado, que circula permanentemente retirando o calor do interior do refrigerador e trocando-o com o exterior. O fluído refrigerante normalmente utilizado nos refrigeradores domésticos caracteriza-se por possuir elevado valor de calor latente de condensação e baixa temperatura de ebulição, além de não ser inflamável. Inicialmente usava-se o gás freon 12 (clorofluorcarbono) como fluído refrigerante, contudo após o mesmo ser identificado como agressor à camada de ozônio, foi substituído por outros fluídos com propriedades semelhantes porém inofensivos à camada de ozônio, como é o caso do HFC-134A.

Figura 4.1: Refrigerador *Forst Free*

Fonte: *Whirlpool Latin America*

Um refrigerador é basicamente composto pelos componentes conforme ilustrado na Figura 4.2, cujas funções são descritas a seguir:

- *Fluido Refrigerante*: Fluido utilizado para o processo de refrigeração, o qual deve possuir baixa pressão de vaporização e alta pressão de condensação;

- *Compressor*: Sua função é puxar o fluido refrigerante sob baixa pressão através da linha de sucção, comprimi-lo e direcioná-lo ao condensador sob alta pressão e temperatura na fase gasosa (vapor super aquecido), fazendo-o assim circular pela tubulação interna do aparelho;

- *Condensador*: Responsável por prover a troca térmica com o ambiente externo, liberando o calor absorvido no evaporador e no compressor durante a compressão. Neste processo o fluido refrigerante passa do estado de vapor super aquecido para o estado líquido sub resfriado a alta pressão;

- *Filtro Secador*: Possui duas funções importantes. A primeira é reter partículas sólidas que em circulação no circuito podem ocasionar obstrução ou danos a partes mecânicas do compressor. A segunda é absorver totalmente a umidade residual do circuito que porventura não tenha sido removida pelo processo de vácuo, evitando danos ao sistema, tais como: corrosão, aumento das pressões e obstrução do tubo capilar por congelamento da umidade;

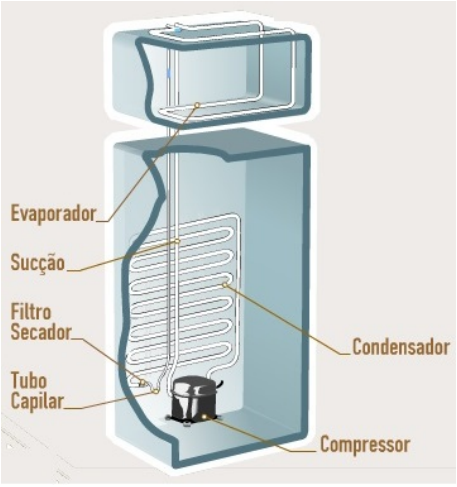
- *Tubo Capilar*: É um tubo de cobre com diâmetro reduzido que age como um elemento de expansão, pois recebe o fluido refrigerante sub resfriado a alta pressão proveniente do condensador e promove uma perda de carga, diminuindo a pressão, separando assim os lados de alta e baixa pressão do circuito;

- *Evaporador*: Composto por um tubo em forma de serpentina, é responsável por receber o fluido refrigerante no estado líquido a baixa pressão e temperatura. Nesta condição, o fluido refrigerante evapora absorvendo o calor da superfície da tubulação do evaporador, ocorrendo

a transformação do líquido sub resfriado para vapor saturado a baixa pressão. Este efeito acarreta a diminuição da temperatura do ambiente interno do refrigerador;

- *Linha de Sucção*: Após absorver o calor ao longo do percurso no evaporador, o fluido refrigerante, retorna ao compressor através da linha de sucção, no estado vapor super aquecido a baixa pressão, para ser então novamente comprimido, dando início a um novo ciclo.

Figura 4.2: Componentes Básicos de um Refrigerador



Fonte: TECUMSEH do Brasil

4.1.2 Tecnologia *Frost Free*

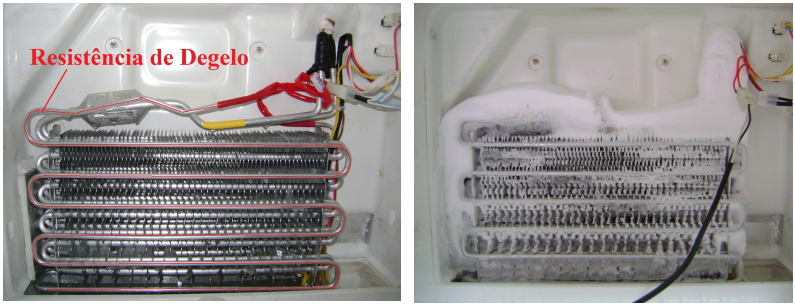
A tecnologia *Frost Free* ou *No Frost* ("sem-gelo", em tradução literal), é um sistema de refrigeração inteligente controlado eletronicamente. É caracterizado pela sua capacidade de refrigeração sem formação de gelo nas paredes do compartimento. Nos refrigeradores comuns,

o evaporador é disposto em torno da parede do *freezer* ou do congelador, fazendo assim com que toda a parede fique bastante fria. A grande desvantagem desta configuração é que toda a umidade do ambiente se condensa sobre essas paredes frias, que em seguida congelam e acabam ficando depositadas nas paredes, ficando cada vez maiores com o passar do tempo diminuindo a eficiência térmica do compartimento.

Já em um refrigerador *Frost Free*, o evaporador fica afastado e termicamente isolado das paredes. Neste sistema somente o ar frio é soprado para dentro do compartimento por um ventilador, mantendo assim este ambiente seco e sem gelo. Contudo, no evaporador, e somente lá, haverá formação de gelo, que por sua vez será periodicamente derretido através da rotina de degelo, que controla o acionamento de uma resistência elétrica montada em torno do evaporador. Na Figura 4.3a é mostrado um evaporador onde é possível visualizar a resistência de degelo já montada em torno do mesmo. Já na Figura 4.3b, é mostrado o mesmo evaporador, contudo já em uma condição crítica de formação de gelo, onde a rotina de degelo precisa ser realizada.

Figura 4.3: (a) Detalhe do Evaporador Montado com uma Resistência de Degelo; (b) Condição do Evaporador com Formação de Gelo

(a) (b)



Fonte: Autor

Devido à necessidade do uso desta resistência elétrica, o consumo de energia neste tipo de refrigerador tende a ser maior em relação aos refrigeradores comuns. Por este motivo, há um grande esforço por parte dos fabricantes em desenvolver refrigeradores que sejam mais eficientes, buscando uma melhor isolação térmica, um controle de temperatura mais robusto e principalmente rotinas de degelo que sejam mais inteligentes, isto é, rotinas que estejam constantemente monitorando o comportamento e uso do refrigerador para assim determinar o melhor momento para realizar o degelo.

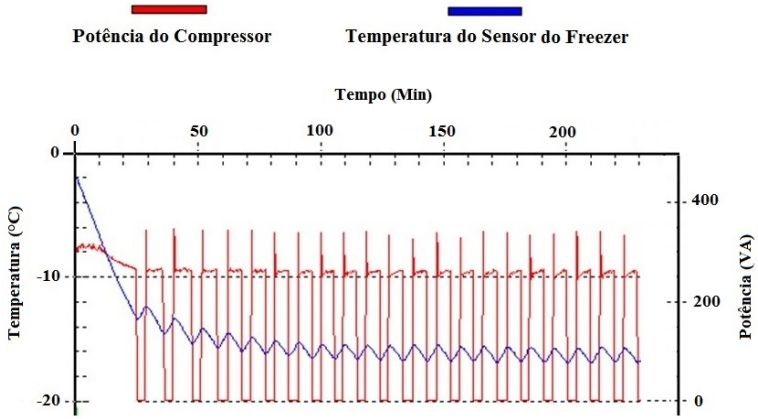
4.1.3 Controle Termostático

O controle de temperatura do tipo de refrigerador escolhido neste estudo de caso baseia-se no liga/desliga do compressor e do ventilador. Através de um termistor do tipo NTC (*Negative Temperature Coefficiente*), é possível monitorar a temperatura do compartimento e assim determinar quando o compressor e o ventilador devem ser ligados ou desligados. Estes valores de temperatura são pré estabelecidos e denominados de T_{ON} (temperatura para ligar o compressor) e T_{OFF} (temperatura para desligar o compressor) e definem uma histerese em torno de um *setpoint*, sendo este o valor médio de temperatura desejado para o compartimento. Na Figura 4.4 é mostrado o gráfico de temperatura e ciclos de acionamentos do compressor, onde é possível verificar como a temperatura varia em torno de T_{ON} e T_{OFF} durante o ciclo termostático.

4.1.4 Rotina de Degelo

A rotina de degelo ou descongelamento é responsável por garantir que o gelo formado no evaporador seja periodicamente derretido, evitando assim que o refrigerador perca eficiência de resfriamento e ainda que este gelo possa migrar para as paredes internas no compartimento. Essa rotina é usualmente formada por duas partes, uma responsável por monitorar as condições de uso e desempenho do produto para assim determinar quando é necessário realizar o degelo, e outra responsável por

Figura 4.4: Forma de Onda da Temperatura Durante os Ciclos Termostáticos



Fonte: Autor

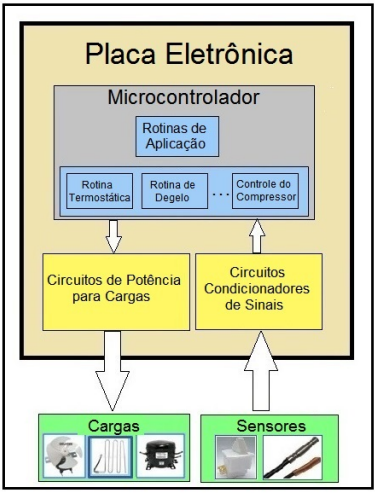
executar o degelo propriamente dito através do controle de acionamento de uma resistência de aquecimento. No entanto, determinar qual o melhor momento para iniciar um degelo não é uma tarefa simples, requer informações tanto dos sensores de temperatura, sensores de abertura de porta e ainda de informações provenientes de algumas rotinas de *software* como fator de funcionamento do compressor, tempo transcorrido desde o último degelo, tempo e número de aberturas de porta, dentre outras. Ou seja, o tempo entre cada degelo pode variar dependendo das condições de uso do produto. Vale a pena ressaltar que intervalos muito longos de degelo podem gerar um acúmulo muito grande de gelo no evaporador, o que prejudica a troca de calor além de prolongar o tempo de resistência ligada para eliminar todo este gelo, aumentando o consumo de energia durante o degelo. Por outro lado, intervalos muito curtos de degelo podem ocasionar uma perda de eficiência do produto, uma vez que ao fim de cada degelo o produto precisa se recuperar, isto é, estabilizar a temperatura em torno do *setpoint*, já que o processo de degelo gera um

calor excessivo que precisa ser retirado. Por razões de segredo industrial não serão mostrados neste trabalho os detalhes do algoritmo responsável por determinar quando o degelo deve ser iniciado, implementado no refrigerador proposto utilizado neste estudo de caso. Será considerado apenas a rotina de controle de acionamento da resistência, já durante o processo de degelo.

4.2 Aplicação do Diagnóstico de Falhas Multicamadas em um Refrigerador Doméstico *Frost Free*

Um refrigerador do tipo *Frost Free* é minimamente composto por uma placa eletrônica onde residem o microcontrolador, circuitos de condicionamento de sinais de sensores, circuitos de potência para acionamento de cargas, dentre outros; por cargas tais como compressor, resistência de aquecimento para fins de degelo e motor do ventilador; e finalmente por sensores, dentre os quais se podem citar os sensores de temperatura (NTC ou PTC por exemplo), sensores de abertura de porta e sinais de *feedback* de circuitos de acionamentos. No microcontrolador encontra-se o *software* que controla tanto as rotinas referentes à aplicação (interação com o usuário) como as rotinas de controle do refrigerador, como por exemplo a rotina termostática, rotina de degelo e a rotina de controle do compressor, conforme mostrado na Figura 4.5

Figura 4.5: Diagrama de Blocos de um Refrigerador



Fonte: Autor

Como mencionado anteriormente, para um diagnóstico efetivo deve-se considerar diagnosticadores para cada uma das três camadas, i.e., deve ser feito o diagnóstico para os elementos que compõem a camada de *software* (rotinas de *software*), a camada de *hardware* (circuitos e componentes elétricos) e a camada de cargas (cargas do sistema). Neste capítulo serão apresentados os detalhes do projeto desses diagnosticadores, conforme mostrado na Figura 4.6, onde para a camada de *software* foi considerado o diagnóstico para a rotina termostática, rotina de degelo e rotina de controle do compressor. Já para a camada de *hardware*, foram admitidos diagnosticadores para os circuitos de acionamento tanto do compressor como da resistência de degelo, circuito este composto por relé e circuito de comando. Finalmente para a camada de cargas, considerou-se diagnosticadores para o compressor e para a resistência de degelo.

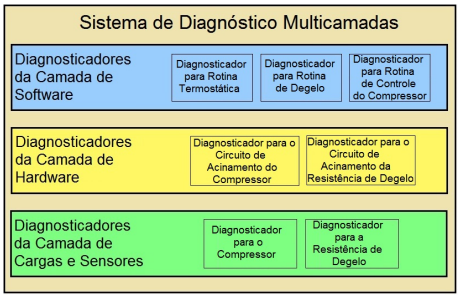
Todos os diagnosticadores foram projetados no contexto de diagnose de falha em sistemas a eventos discretos, conforme proposto por Sampath et al. (1996), e, uma vez que todos os diagnosticadores possuem acesso a todos os eventos observáveis necessários para a diagnose, adotou-se a arquitetura de diagnose centralizada. As secções seguintes detalham o projeto de cada um dos diagnosticador acima mencionados.

4.3 Projeto dos Diagnosticadores da Camada de Software

Na camada de *software* há três rotinas cujo diagnóstico é necessário: a rotina termostática, a rotina de controle do compressor e a rotina de degelo, descritas a seguir. Definimos então o conjunto de falhas pertencente à camada de *software*, como segue:
$\Sigma_{f_{sw}} = \{\sigma_{sw_1}, \sigma_{sw_2}, \sigma_{sw_3}\}$
onde:

- $\sigma_{sw_1} = TC\_Fail \Rightarrow$ Evento de falha da rotina termostática;

Figura 4.6: Sistema de Diagnóstico Multicamadas para um Refrigerador



Fonte: Autor

- $\sigma_{sw_2} = Comp\_Ctrl\_Fail \Rightarrow$ Evento de falha da rotina de controle do compressor;

- $\sigma_{sw_3} = Def\_Fail \Rightarrow$ Evento de falha da rotina de degelo.

Uma vez que temos acesso a algumas informações do *software* de controle, consideraremos essas informações como se fossem provenientes de sensores, ou seja, teremos uma espécie de "sensor virtual", cuja informação é simplesmente um dado da memória referente a alguma variável interna ao *software*.

4.3.1 Diagnosticador da Rotina Termostática

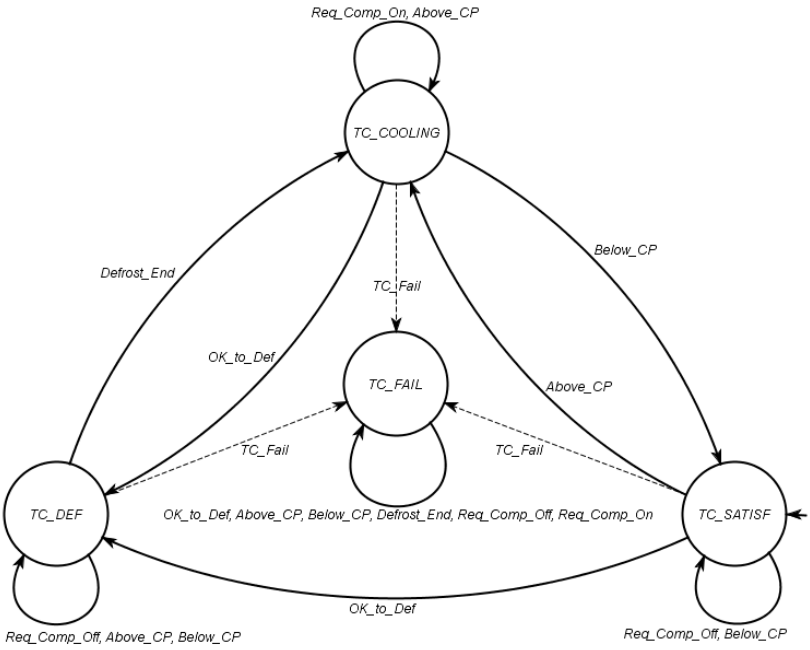
A rotina termostática é responsável por controlar quando o compressor deve ser ligado e desligado para assim manter a temperatura do refrigerador dentro da faixa de temperatura selecionada (*setpoint*). Para efetuar este controle, esta rotina se baseia na leitura proveniente do sensor de temperatura, comparando-a com o valor de referência (T_{ON}) e (T_{OFF}), conforme o *setpoint* selecionado. Assim, se a temperatura está acima do valor de referência para acionamento do compressor (T_{ON}), esta rotina deve requisitar que o compressor seja ligado, e quanto a tempera-

tura estiver abaixo do valor de referência de desligamento (T_{OFF}), esta rotina deve então solicitar o desligamento do compressor.

O autômato que representa o modelo da rotina termostática G_{TC} é mostrado na Figura 4.7. Esta rotina é composta por três estados principais, aqui denominados como *TC_SATISF*, *TC_COOLING* e *TC_DEF*. O quarto estado *TC_FAIL* foi adicionado para representar o estado de falha desta rotina. O estado *TC_SATISF* (Termostática Satisfeita) representa a condição onde a rotina termostática está satisfeita, isto é, a temperatura está abaixo de T_{OFF} (evento *Below_CP*) e portanto enquanto esta condição perdurar a rotina termostática deve requisitar que o compressor permaneça desligado, condição esta representada pelo evento *Req_Comp_Off*. No entanto, caso a temperatura suba acima de T_{ON}, o evento *Above_CP* é gerado, fazendo a rotina mudar então para o estado *TC_COOLING* (Termostática Resfriando), e requisitar assim que o compressor seja ligado, através do evento *Req_Comp_On* e esta condição deve se manter enquanto a temperatura estiver acima do valor de desligamento do compressor (T_{OFF}). Agora, a partir do estado *TC_COOLING*, se a temperatura cair abaixo de T_{OFF}, o evento *Below_CP* é gerado fazendo a rotina termostática retornar ao estado *TC_SATISF*. Para ambos estados *TC_SATISF* e *TC_COOLING*, caso um processo de degelo precise ser iniciado, o evento *OK_to_Def* (OK para Degelo) será gerado e a termostática irá então mudar para o estado *TC_DEF* (Termostática Degelando) e o compressor deve ser requisitado para ser mantido desligado, mesmo que a temperatura suba acima da temperatura de ligamento. Esta condição é representada pelos auto-laços *Req_Comp_Off*, *Above_CP* e *Below_CP*. Uma vez no estado *TC_DEF*, após a ocorrência do evento de fim de degelo (*Defrost_End*), a rotina retornará ao estado *TC_COOLING*. Em condições normais é esperado que após o fim de um degelo a temperatura esteja acima do *setpoint*, contudo por questão de simplicidade independentemente da temperatura, ao término de um degelo a rotina sempre retorna ao estado *TC_COOLING*. Os autolaços *Req_Comp_On* no estado *TC_COOLING* e *Req_Comp_Off* nos estados *TC_SATISF* e *TC_DEF* são usados para representar o fato de

que essa rotina faz uma atualização periódica da condição requisitada do compressor (ligado ou desligado). De maneira análoga, as condições de temperatura *Above_CP* e *Below_CP* também são periodicamente atualizados. A falha desta rotina é representada pelo evento não observável *TC_Fail* (Falha na Termostática). Na ocorrência deste evento, a partir de qualquer estado (*TC_SATISF*, *TC_COOLING* ou *TC_DEF*) o autômato mudará para o estado *TC_FAIL* e ali ficará preso.

Figura 4.7: Autômato que Modela a Rotina Termostática, G_{TC}



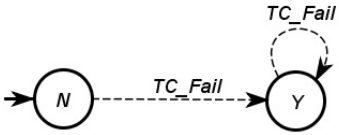
Fonte: Autor

Observe que em nenhum momento a rotina termostática liga ou desliga diretamente o compressor, ela apenas faz requisições para ligar ou desligar o mesmo. Estas requisições são recebidas pela rotina de con-

trole do compressor e esta por sua vez é quem irá atuar para efetivamente ligar ou desligar o compressor. Esta rotina será detalhada mais adiante na seção 4.3.2.

A Figura 4.8 mostra a autômato rotulador Al_{Th} considerado para o diagnóstico da rotina termostática.

Figura 4.8: Autômato Rotulador para Falha na Rotina Termostática, Al_{Th}

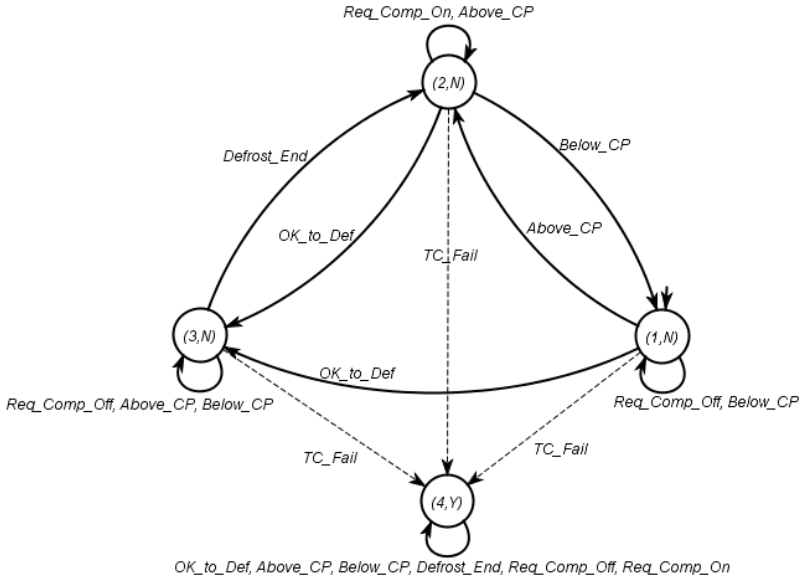


Fonte: Autor

O próximo passo para se obter o diagnosticador Gd_{Th} é determinar a composição síncrona da planta G_{TC} com o autômato rotulador Al_{Th}, isto é fazer $(G_{TC}||Al_{Th})$, cujo resultado é mostrado na Figura 4.9.

Em seguida, o diagnosticador Gd_{Th} é obtido determinando-se o observador deste último autômato, ou seja, $Gd_{Th} = Obs(G_{TC}||Al_{Th})$, mostrado da Figura 4.10. O estado $\{4Y\}$ representa o estado onde o diagnosticador está certo da ocorrência da falha. Uma vez atingido este estado, não há como o diagnosticador retornar para algum estado incerto, ou seja, ele ficará ali preso independente de qual evento ocorra, satisfazendo assim a condição de diagnosticabilidade enunciada por Sampath et al. (1995) e Basilio et al. (2010). É importante ressaltar que embora o diagnosticador obtido possua ciclos formados apenas por estados incertos, isso não necessariamente implica na impossibilidade de se diagnosticar a ocorrência de uma falha. Conforme mostrado anteriormente na seção 2.8, para que L seja diagnosticável em relação a P_o e Σ_f, após a ocorrência da falha, não pode existir um ciclo de estados em G que corresponda a um ciclo de estados incertos no diagnosticador. Um exemplo

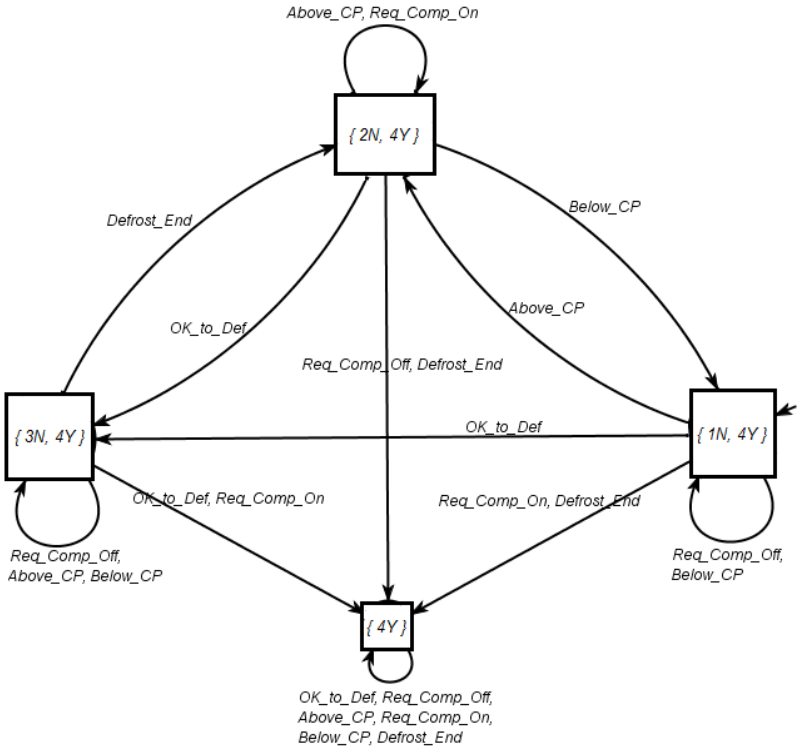
Figura 4.9: Autômato obtido pela composição $G_{TC}||Al_{Th}$



Fonte: Autor

que ilustra a situação onde um ciclo de estados incertos não formam um ciclo indeterminado no diagnosticador pode ser encontrado em Cassandras e Lafortune (2008, p.114). Como pode-se observar na Figura 4.7, após ocorrência da falha independente do estado atual, o autômato sempre alcançará para o estado de falha, isso não deixará o diagnosticador preso no ciclos formados apenas por estados incertos.

Figura 4.10: Diagnosticador para Rotina Termostática, Gd_{Th}



Fonte: Autor

4.3.2 Diagnosticador da Rotina de Controle do Compressor

Essa rotina gerencia as requisições para ligar e desligar o compressor. Ela irá avaliar e tratar adequadamente estas requisições antes de atuar no circuito de acionamento para assim ligar ou desligar o compressor, conforme solicitado.

Como mostrado na Figura 4.11, o autômato $G_{CompRoutine}$ que modela o comportamento desta rotina possui sete estados, como segue:

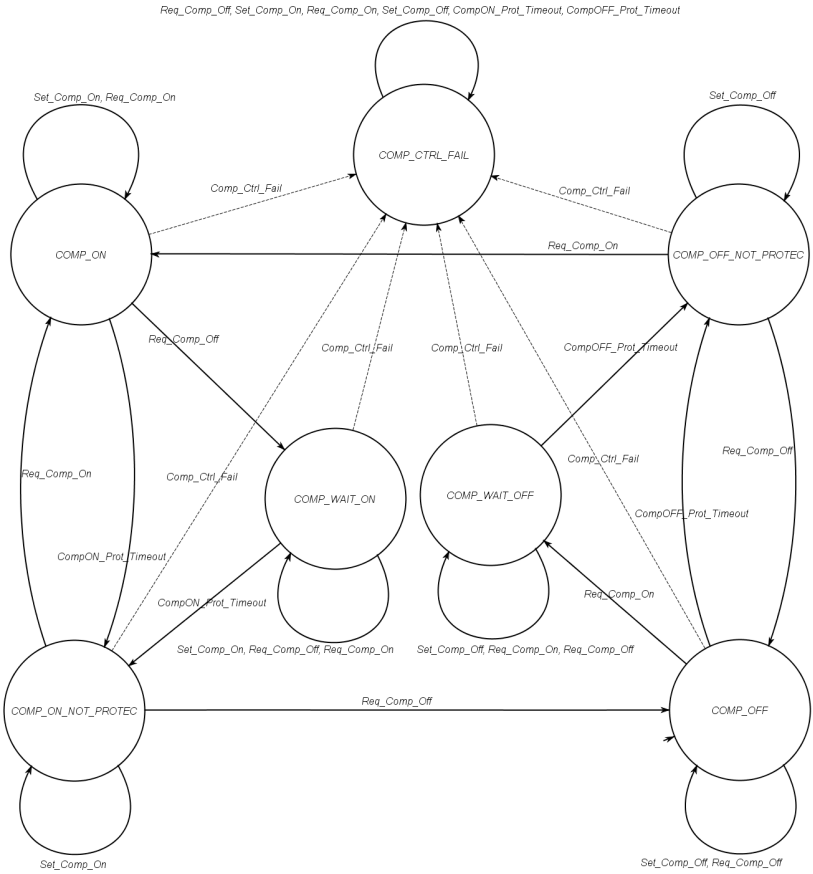
• *COMP_ON* (Compressor Ligado);
• *COMP_OFF* (Compressor Desligado);
• *COMP_OFF_NOT_PROTEC* (Compressor Desligado e não protegido);
• *COMP_ON_NOT_PROTEC* (Compressor Ligado e não protegido);
• *COMP_WAIT_ON* (Compressor Ligado e protegido, após requisição para desligar),
• *COMP_WAIT_OFF* (Compressor Desligado e protegido, após requisição para ligar) e
• *COMP_CTRL_FAIL* (Falha da Rotina de Controle do Compressor),

Este autômato possui o seguinte conjunto de eventos:
- *Req_Comp_On* (Requisição para Ligar o Compressor);
- *Req_Comp_Off* (Requisição para Desligar o Compressor);
- *Set_Comp_On* (Liga Compressor);
- *Set_Comp_Off* (Desliga Compressor);
- *CompON_Prot_Timeout* (Fim da Proteção de Compressor Ligado) e
- *CompOFF_Prot_Timeout* (Fim da Proteção de Compressor Desligado).

O estado *COMP_CTRL_FAIL* representa o estado de falha da rotina de controle do compressor.

O modelo desta rotina considera duas proteções, a primeira está relacionada com o tempo mínimo que o compressor deve ser mantido desligado e a segunda refere-se ao tempo mínimo na qual o compressor deve ser mantido ligado. O objetivo da primeira proteção é garantir que o compressor ficará desligado o mínimo de tempo necessário para equalizar a pressão do sistema de refrigeração, caso contrário, um sobreaquecimento ocorrerá sob as bobinas do motor do compressor, podendo danificá-lo. Já e segunda proteção é usada para os casos onde se deseje fixar um tempo mínimo de funcionamento do compressor entre cada ciclo de liga e desliga do compressor, e assim melhor controlar o fator de funcionamento, que é um dos parâmetros essenciais para a tomada de decisão de quando iniciar um novo ciclo de degelo.

Figura 4.11: Autômato que Modela a Rotina de Controle do Compressor, $G_{CompRoutine}$



Fonte: Autor

Assim, caso o compressor esteja ligado (estado *COMP_ON*) e chegue uma requisição para desligá-lo enquanto a proteção de tempo mínimo ligado ainda não expirou, a rotina estará ciente desta nova re-

quisição, mudando para o estado *COMP_WAIT_ON*, mantendo ainda o compressor ligado até que o tempo expire, quando mudará então para o estado *COMP_ON_NOT_PROTEC*. Neste momento, se a requisição para desligar o compressor ainda persistir a rotina mudará para o estado *COMP_OFF*, e finalmente o compressor será desligado. Entretanto, caso a proteção de tempo mínimo ligado expire antes da ocorrência da requisição de desligar, a rotina irá para o estado *COMP_ON_NOT_PROTEC*, mantendo o compressor ainda ligado. Uma vez neste estado, ocorrendo agora sim, a requisição para desligar, a rotina atenderá essa requisição sem restrições, mudando para o estado *COMP_OFF* desligando assim o compressor.

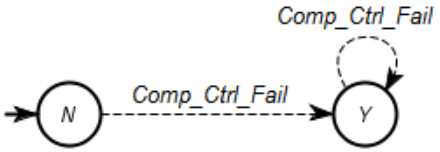
De maneira similar, quando o compressor está desligado, porém ainda dentro do tempo de proteção de tempo mínimo desligado, e há uma requisição para ligar, o autômato mudará para o estado *COMP_WAIT_OFF*, indicando que a rotina está ciente desta requisição mas manterá o compressor desligado até que o tempo de proteção expire. Quando isso acontecer, o autômato irá para o estado *COMP_OFF_NOT_PROTEC*, e se a requisição de ligar o compressor ainda se mantiver, ele mudará para o estado *COMP_ON*, ligando finalmente o compressor. Caso contrário, se a proteção expirar sem que tenha ocorrido alguma requisição para ligar o compressor, o autômato irá para o estado *COMP_OFF_NOT_PROTEC*. Neste estado, ocorrendo uma requisição para ligar, a mesma será atendida e o autômato irá para o estado *COMP_ON*.

A proteção de tempo mínimo ligado é ativada sempre que há uma transição do estado do compressor de desligado para ligado e de maneira análoga, a proteção de tempo mínimo desligado é ativada sempre que há uma transição do compressor do estado ligado para o estado desligado. No entanto estes eventos de ativação não são considerados no modelo pois não são relevantes do ponto de vista de diagnóstico. Somente os eventos relacionados à expiração destas proteções se mostraram relevantes para o diagnóstico, e são representadas, conforme mencionado,

pelos eventos *CompON_Prot_Timeout* e *CompOFF_Prot_Timeout* respectivamente.

Na Figura 4.12 mostra-se o autômato rotulador $Al_{CompModel}$ usado para o diagnóstico da rotina de controle do compressor.

Figura 4.12: Autômato Rotulador para Falha na Rotina de Controle do Compressor, $Al_{CompModel}$



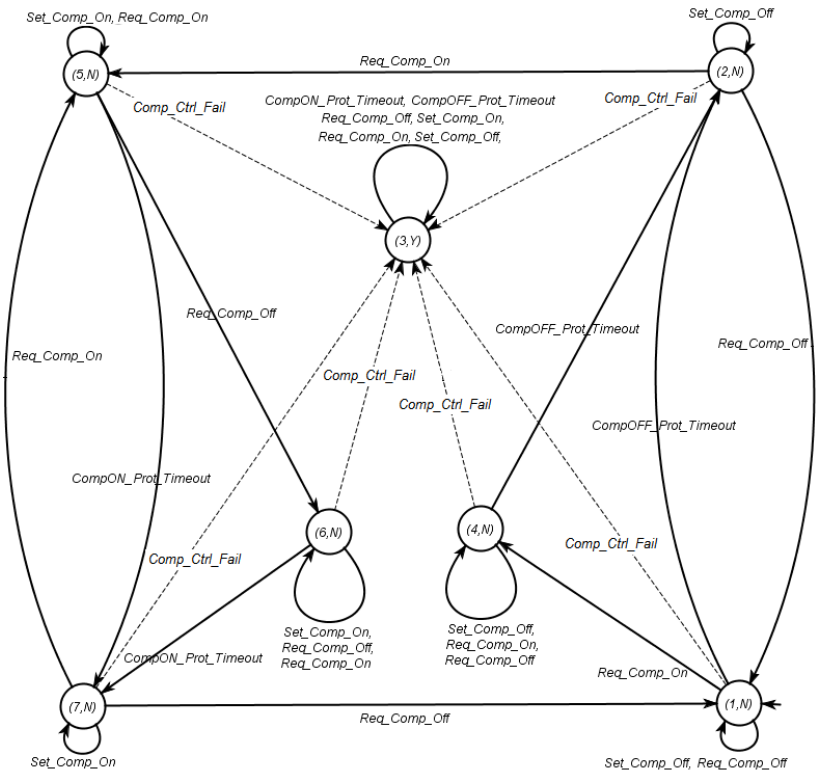
Fonte: Autor

O próximo passo para obter o diagnosticador $Gd_{CompRoutine}$ é determinar a composição síncrona de $G_{CompRoutine}$ com Al_{Comp}, cujo autômato resultante é mostrado na Figura 4.13. Por fim, o diagnosticador da rotina de controle do compressor é obtido determinando-se o observador deste autômato, ou seja, $Gd_{CompRoutine} = Obs(G_{CompRoutine}||Al_{CompModel})$ (ver Figura 4.14) .

Por questões de espaço, no intuito de obter uma figura legível, para a representação do diagnosticador, os nomes originais dos eventos foram substituídos por identificadores (IDs) de uma única letra, cuja correspondência é mostrada na Tabela 4.1. O estado $\{3Y\}$ representa a situação em que o diagnosticador está certo da ocorrência da falha, e uma vez alcançado este estado, o diagnosticador ali ficará preso, não sendo possível retornar para um estado incerto.

De maneira análoga ao diagnosticador obtido para a rotina termostática na seção anterior, o fato do diagnosticador obtido possuir ciclos formados apenas por estados incertos, não implica na impossibilidade de se diagnosticar a ocorrência da falha. Para que L seja diagnosticável em

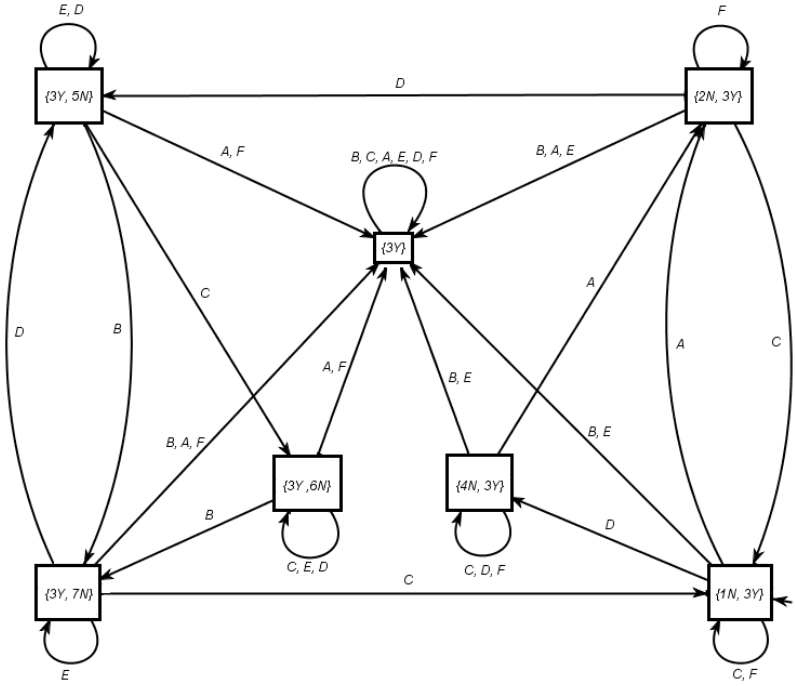
Figura 4.13: Autômato resultante de $G_{CompRoutine}||Al_{CompModel}$



Fonte: Autor

relação a P_o e Σ_f, não pode existir um ciclo de estados em G após a ocorrência da falha que corresponda ao ciclo de estados incertos no diagnosticador. Como se pode observar na Figura 4.11, após ocorrência da falha, independentemente do estado atual, o autômato sempre alcançará o estado de falha. Isso não deixará o diagnosticador preso no ciclos formados apenas por estados incertos.

Figura 4.14: Diagnosticador para a Rotina de Controle do Compressor, $Gd_{CompRoutine}$



Fonte: Autor

4.3.3 Diagnosticador da Rotina de Degelo

A rotina de degelo é uma rotina extremamente crítica para um refrigerador do tipo *Frost Free*, pois é ela quem controla a operação da resistência de degelo para garantir um degelo automático e eficiente, o que caracteriza um refrigerador do tipo *Frost Free*. Essa rotina é com-

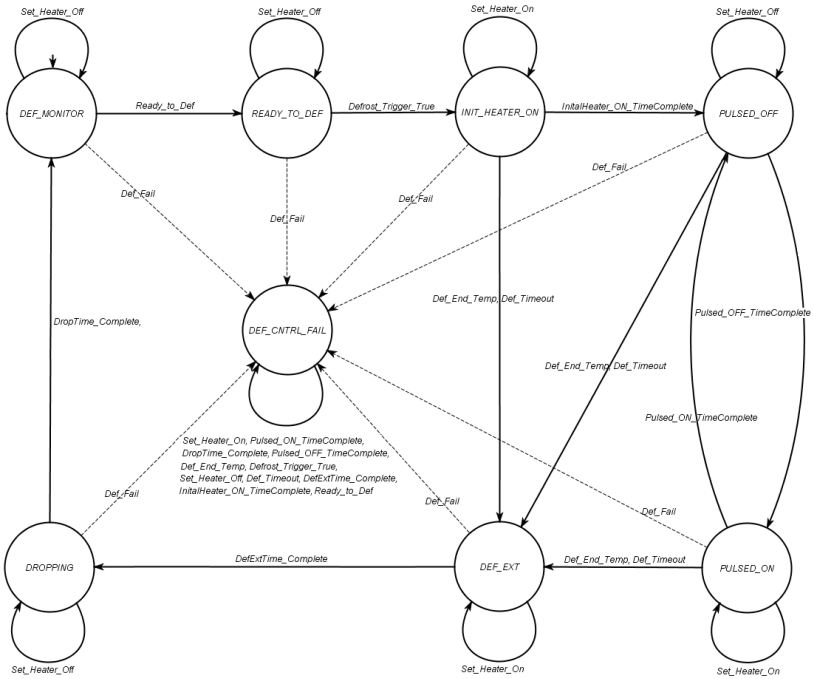
Tabela 4.1: Tabela de Eventos para o Diagnosticador da Rotina do Compressor

Nome do Evento	ID do Evento
$CompOff\_Prot\_Timeout$	A
$CompOn\_Prot\_Timeout$	B
Req_Comp_Off	C
Req_Comp_On	D
Set_Comp_On	E
Set_Comp_Off	F

posta por diversos passos que vão desde monitorar o comportamento do refrigerador, coletando e processando informações para determinar o melhor momento para iniciar um novo ciclo de degelo até realizar o controle de ativação da resistência de degelo durante a execução do ciclo de degelo. O autômato que modela o algoritmo de *software* desta rotina, é mostrado a seguir na Figura 4.15.

O primeiro estado desta rotina é o *DEF_MONITOR* (Monitor de Degelo). Neste passo a rotina está aguardando que as condições de início de degelo sejam satisfeitas. Por questões de sigilo industrial, sem que haja prejuízo para o modelo de diagnóstico aqui proposto, essas condições não serão mostradas. Quando estas condições são satisfeitas, o evento *Ready_to_Def* (Pronto para Degelo) é gerado, levando assim o autômato para o estado *READY_TO_DEF* (Pronto para Degelo). Neste estado, a rotina está esperando pelo evento de confirmação de início de degelo *Defrost_Trigger_True* (Disparo de Degelo Verdadeiro), para assim iniciar o processo de degelo. Essa confirmação é feita pela rotina de aplicação (rotina responsável por gerenciar as configurações selecionadas pelo usuário, tais como nível de temperatura, rotinas de congelamento rápido, modo férias, turbo gelo, etc.), pois em alguns casos, ela pode ser postergada devido à execução de alguma outra rotina que não pode ser interrompida no momento, como por exemplo as rotinas de congelamento rápido e turbo gelo. Assim que o início de degelo é confirmado (evento *Defrost_Trigger_True* é gerado), o autômato mudará então para

Figura 4.15: Autômato que Modela a Rotina de Degelo, $G_{DefModel}$



Fonte: Autor

o próximo estado, denominado *INIT_HEATER_ON* (Estado Inicial de Resistência Ligada), onde a resistência de degelo é ligada. Neste momento, a rotina passa a verificar três condições, representadas pelos eventos abaixo listados:

i) *Initial_HeaterON_TimeComplete*: Tempo inicial de resistência ligada completo;

ii) *Def_End_Temp*: Temperatura de fim de degelo alcançada, e

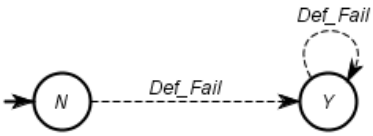
iii) *Def_Timeout*: Tempo máximo de degelo alcançado.

O primeiro evento refere-se ao máximo tempo permitido para a resistência permanecer ligada neste passo, o segundo se a temperatura de fim de degelo foi atingida e o terceiro se o tempo máximo de execução da rotina de degelo foi alcançado.

Na ocorrência do evento *Initial_HeaterON_TimeComplete*, a rotina de degelo irá então alternar a resistência entre desligada e ligada, através da alternância dos estados *PULSED_OFF* (Pulso Inativo) e *PULSED_ON* (Pulso Ativo) respectivamente, através da ocorrência dos eventos *Pulsed_OFF_TimeComplete* (Tempo de Pulso Ativo Completo) e *Pulsed_ON_TimeComplete* (Tempo de Pulso Inativo Completo) respectivamente. Quando a temperatura de fim de degelo ou o tempo máximo de duração do gelelo são atingidos, ou seja, na ocorrência do evento *Def_End_Temp* ou *Def_Timeout* respectivamente, o próximo estado permitido é o *DEF_EXT* (Degelo Estendido). Neste estado, a resistência pode ser mantida ligada por um tempo extra, se necessário, ou alguma resistência auxiliar pode também ser ligada, até que o evento *DefExtTime_Complete* (Tempo Extra de Degelo Completo) ocorra, levando assim a rotina para o último estado, denominado *DROPPING* (Escorrimento). O objetivo deste último estado é apenas esperar até que toda a água proveniente do degelo escorra pelo duto de saída antes de ligar novamente o compressor, evitando assim que esta água congele dentro dos dutos, bloqueando-os. Assim que este tempo expirar, o evento *DropTime_Complete* (Tempo de Escorrimento Completo) será gerado levando então o autômato para o estado inicial *DEF_MONITOR* novamente.

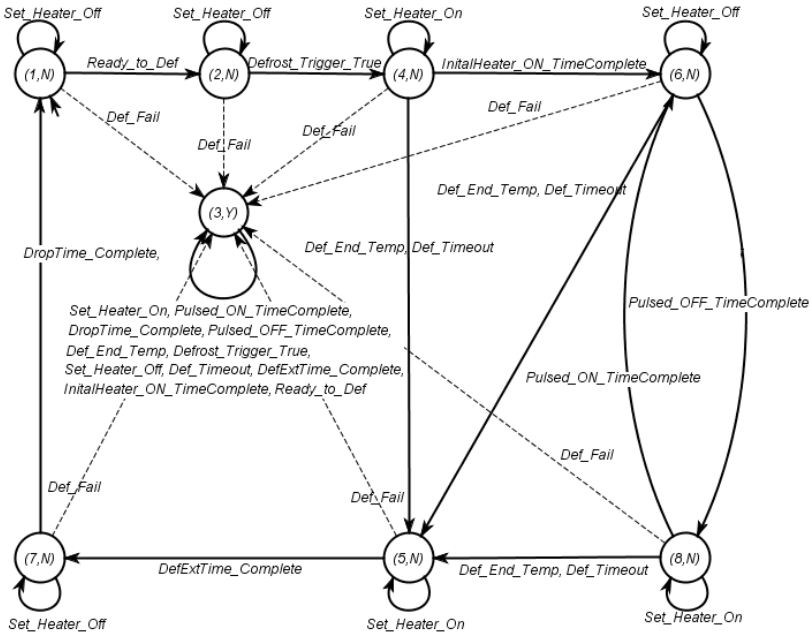
O estado *DEF_CNTRL_FAIL* (Falha na Rotina de Degelo) representa o estado de falha desta rotina devido à ocorrência do evento não observável *Def_Fail* (Falha no Degelo). A Figura 4.16 mostra o autômato rotulador $Al_{DefLabel}$ para esta rotina e a Figura 4.17 a composição síncrona entre o modelo da rotina de degelo e seu autômato rotulador, isto é $G_{DefModel}||Al_{DefLabel}$.

Figura 4.16: Autômato Rotulador para Falha na Rotina de Degelo, $Al_{DefLabel}$



Fonte: Autor

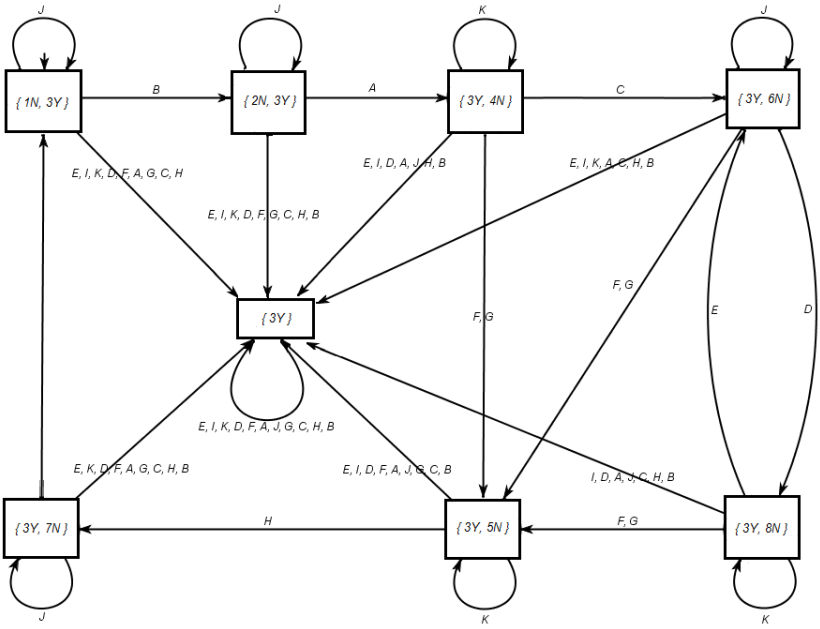
Figura 4.17: Autômato obtido pela composição $G_{DefModel}||Al_{DefLabel}$



Fonte: Autor

O diagnosticador $Gd_{DefModel}$ foi obtido calculando-se o observador deste último autômato, isto é, $Gd_{DefModel} = Obs(G_{DefModel}||Al_{DefModel})$, mostrado da Figura 4.18

Figura 4.18: Diagnosticador para a Rotina de Degelo, $Gd_{DefModel}$



Fonte: Autor

Assim como feito na seção anterior, no intuito de obter uma figura legível para representação do diagnosticador, os nomes originais dos eventos foram substituídos por identificadores (IDs) de uma única letra, cuja correspondência é mostrada na Tabela 4.2. O estado $\{3Y\}$ representa a situação em que o diagnosticador está certo da ocorrência da falha, e uma vez alcançado este estado, o diagnosticador ali ficará preso, não sendo possível retornar para um estado incerto.

De maneira análoga aos diagnosticadores obtidos nas seções anteriores, a condição de diagnosticabilidade não foi violada, uma vez que o diagnosticador não possui ciclos indeterminados, isto é, após a ocorrência da falha o diagnosticador consegue sair deste ciclo de estados incertos. Como pode-se observar na Figura 4.15, após ocorrência da falha independente do estado atual, o autômato sempre alcançará para o estado de falha, isso não deixará o diagnosticador preso no ciclos formados apenas por estados incertos.

Tabela 4.2: Tabela de Eventos para o Diagnosticador da Rotina de Degelo

Nome do Evento	**ID do Evento**
$Defrost\_Trigger\_True$	A
$Ready\_to\_Def$	B
$InitalHeater\_ON\_TimeComplete$	C
$Pulsed\_OFF\_TimeComplete$	D
$Pulsed\_ON\_TimeComplete$	E
Def_End_Temp	F
$Def\_Timeout$	G
$DefExtTime\_Complete$	H
$DropTime\_Complete$	I
Set_Heater_Off	J
Set_Heater_On	K

4.4 Projeto dos Diagnosticadores da Camada de Hardware

Na camada de *hardware* existem dois circuitos importantes para os quais o diagnóstico deve ser considerado: circuito de acionamento do compressor e circuito de acionamento da resistência de degelo. Conforme mencionado anteriormente, ambos são compostos por um circuito de comando do relé e pelo próprio relé. Para fins de diagnóstico, tanto a parte referente ao comando como o relé serão considerados como um dispositivo único e daqui por diante serão referidos apenas por relés.

Definimos então o conjunto de falhas pertencente à camada de *hardware*, como segue:
$\Sigma_{f_{hw}} = \{\sigma_{hw_1}, \sigma_{hw_2}, \sigma_{hw_3}, \sigma_{hw_4}\}$
onde:

- $\sigma_{hw_1} = FailOpen \Rightarrow$ Evento de falha para o relé do compressor travado aberto;
- $\sigma_{hw_2} = FailClosed \Rightarrow$ Evento de falha para o relé do compressor travado fechado;
- $\sigma_{hw_3} = FailOpen \Rightarrow$ Evento de falha para o relé da resistência de degelo travado aberto;
- $\sigma_{hw_4} = FailClosed \Rightarrow$ Evento de falha para o relé da resistência de degelo travado fechado;

4.4.1 Diagnosticadores dos Relés

A placa eletrônica usada neste projeto possui um circuito de *feedback*, o qual faz uma amostragem do sinal na saída do relé. Este *feedback* pode então ser usado como um sensor de resposta de acionamento do relé, podendo assim indicar o estado do relé, aberto ou fechado. Assim, quando o relé abre e, portanto não há nenhum sinal presente em sua saída, é gerado o evento FB_Off (*feedback* de relé desligado), de modo análogo, quando o relé fecha, é gerado o evento FB_On (*feedback* de

relé ligado). As leituras provenientes deste sensor serão consideradas no projeto dos diagnosticadores.

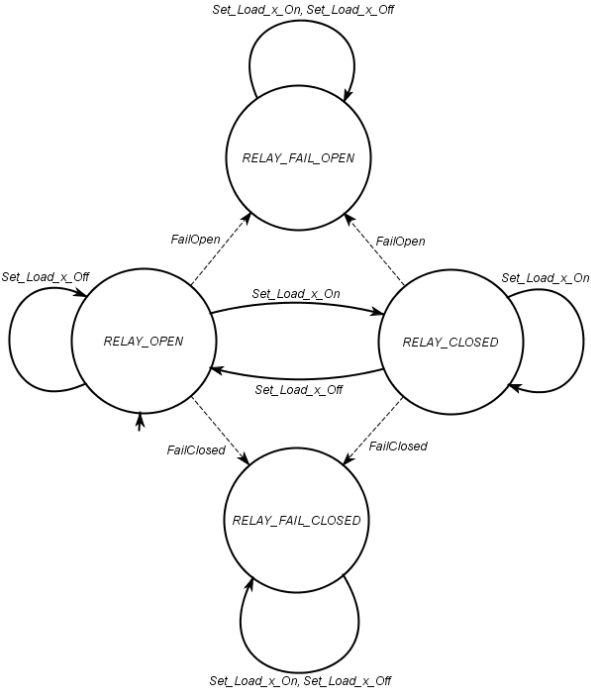
Uma vez que ambos os relés (compressor e resistência de degelo) possuem o mesmo modelo, por simplicidade será usado um modelo e um diagnosticador genérico para ambos relés, onde na notação *Load_x*, o *x* deve ser entendido como *Compressor* ou *Resistência de Degelo* para cada caso. Na Figura 4.19 mostra-se o modelo para os relés e, como se pode notar, diferentemente dos casos anteriores, este modelo considera dois tipos de falhas para o relé: *FailOpen* (Falha de Relé Travado Aberto) e *FailClosed* (Falha de Relé Travado Aberto).

Contudo este modelo não possui um conjunto de eventos adequado para o diagnóstico, pois conforme mostrado na Figura 4.20, o diagnosticador possui apenas estados incertos, ou seja, com base nos eventos no modelo proposto na Figura 4.19 a ocorrência das falhas não é diagnosticável. Assim, um mapeamento de sensor (Sampath et al., 1996) se faz necessário neste caso. O mapeamento de sensor basicamente consiste em uma tabela que relaciona os estados dos sensores com cada estado do autômato. Usando as informações de *feedback* dos relés, foi definido o mapeamento de sensor mostrado na Tabela 4.3.

Neste mapeamento, cada estado do relé foi relacionado com os comandos para ligar e desligar o relé ($Set\_Load\_x\_On$ e $Set\_Load\_x\_Off$, respectivamente) e seu respectivo *feedback* (FB_On e FB_Off). Assim, para o estado $RELAY\_OPEN$ (relé aberto) por exemplo, tem-se os eventos $Set\_Load\_x\_Off$ e $FB\_Off]$, que indicam que o relé foi requisitado para ser desligado e seu respectivo sinal de *feedback* detectou que o mesmo desligou. De modo análogo no estado $RELAY\_FAIL\_CLOSED$ (relé em falha fechado), tem-se a situação onde o relé foi requisitado para desligar através do evento $Set\_Load\_x\_Off$, contudo seu respectivo *feedback* indicou que o mesmo permaneceu ligado gerando então o evento FB_On.

O autômato que modela o comportamento dos relés considerando o mapeamento de sensores pode ser visto na Figura 4.21.

Figura 4.19: Autômato que Modela os Relés, $G_{Load\_x\_Relay}$

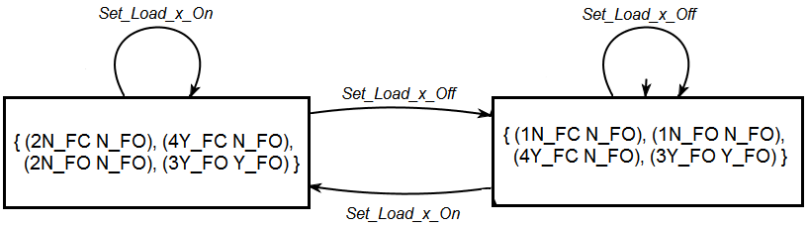


Fonte: Autor

Os autômatos rotuladores tanto para a falha *FailOpen* como para a falha *FailClosed* são mostrado a seguir na Figura 4.22.

O autômato resultante da composição síncrona do modelo do relé com ambos autômatos rotuladores pode ser visto na Figura 4.23 e seu respectivo diagnosticador $Gd_{Load\_x} = Obs(G_{Map\_Load\_x\_Relay} \mid\mid Al_{RelayClosed} \mid\mid Al_{RelayOpen})$ na Figura 4.24.

Figura 4.20: Diagnosticador para o Modelo $G_{Load\_x\_Relay}$



Fonte: Autor

Tabela 4.3: Mapeamento de Sensor para o Modelo dos Relés

Estado	**Mapeamento**
$RELAY\_OPEN$	$[Set\_Load\_x\_Off, FB\_Off]$
$RELAY\_CLOSED$	$[Set\_Load\_x\_On, FB\_On]$
$RELAY\_FAIL\_OPEN$	$[Set\_Load\_x\_On, FB\_Off]$
$RELAY\_FAIL\_CLOSED$	$[Set\_Load\_x\_Off, FB\_On]$

4.5 Projeto dos Diagnosticadores da Camada de Cargas

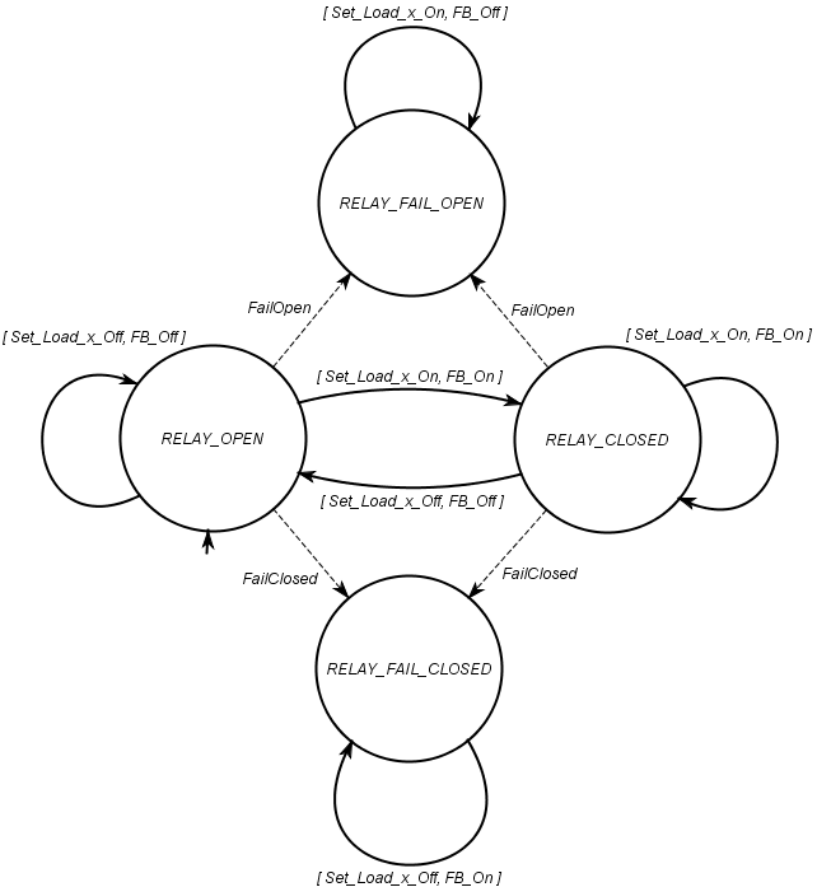
Esta seção apresenta o projeto dos diagnosticadores para as duas cargas pertencentes a esta camada: o compressor e a resistência de degelo.

Definimos então o conjunto de falhas pertencente à camada de cargas, como:
$\Sigma_{f_{cs}} = \{\sigma_{cs_1}, \sigma_{cs_2}\}$
onde:

- $\sigma_{cs_1} = Comp\_Fail \Rightarrow$ Evento de falha do compressor;

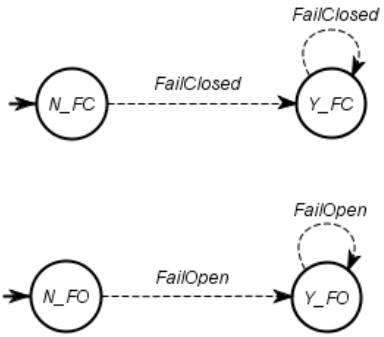
- $\sigma_{cs_2} = Heater\_Fail \Rightarrow$ Evento de falha da resistência de degelo;

Figura 4.21: Autômato que Modela os Relés considerando Mapeamento de Sensor, $G_{Map\_Load\_x\_Relay}$



Fonte: Autor

Figura 4.22: Autômatos Rotuladores para Falhas de Relé, $Al_{RelayClosed}$ e $Al_{RelayOpen}$



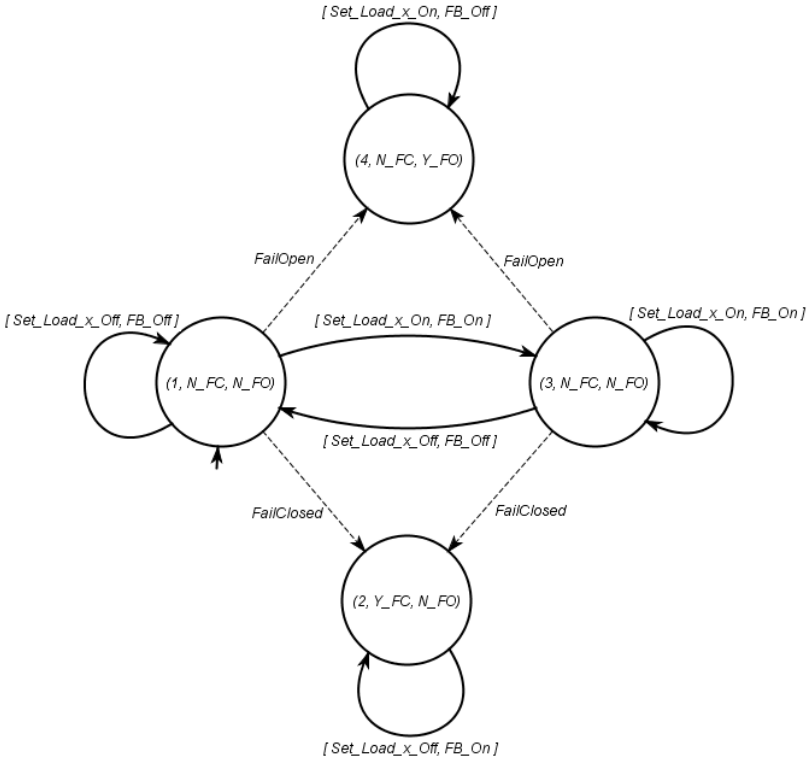
Fonte: Autor

4.5.1 Diagnosticador para o Compressor

O autômato que modela o funcionamento do compressor é mostrado na Figura 4.25 e é composto pelos seguintes estados: *C_ON* (Compressor Ligado), *C_OFF* (Compressor Desligado) e pelo estado de falha *CS_OFF* (Compressor Travado Desligado).

Note que a falha do tipo *Travado Ligado* não é considerada neste modelo, pois do ponto de vista do compressor (motor), esta falha implicaria em ter o compressor sempre ligado mesmo quando não há tensão sobre ele (quando o relé que o aciona está desligado), o que não é fisicamente possível de ocorrer. Entretanto, uma falha no circuito de acionamento ou em alguma rotina de *software* pode fazer com que o compressor fique constantemente ligado. Caso isso ocorra, a origem da falha não será o compressor mas sim em algum dos elementos das camadas superiores, isto é, na camada de *hardware* para o caso de falha de relé travado fechado, ou na camada *software* no caso de uma falha na rotina termostática ou de controle do compressor por exemplo. Assim, os di-

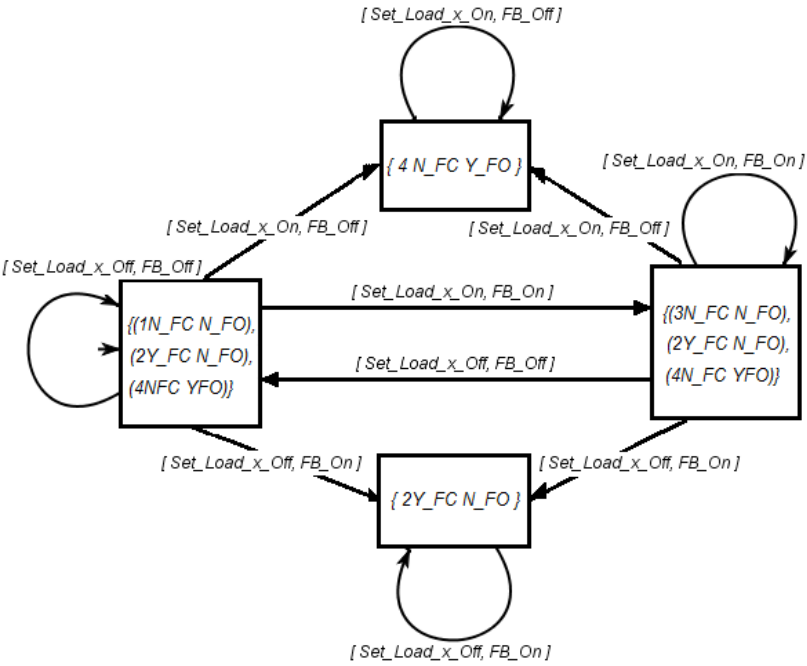
Figura 4.23: Autômato obtido pela composição $G_{Map\_Load\_x\_Relay}$ || $Al_{RelayClosed}$ || $Al_{RelayOpen}$



Fonte: Autor

agnosticadores pertencentes a estas camadas deverão detectar a falha. Contudo, no caso da falha *Compressor Travado Desligado*, o compressor permanece desligado mesmo quando alimentado com sua tensão de funcionamento, o que indica que a falha está realmente no compressor e não em qualquer outro elemento das camadas superiores. Normalmente esta falha está relacionada a danos físicos do compressor, como por exemplo,

Figura 4.24: Diagnosticador para os Relés, $Gd_{Load\_x\_Relay}$



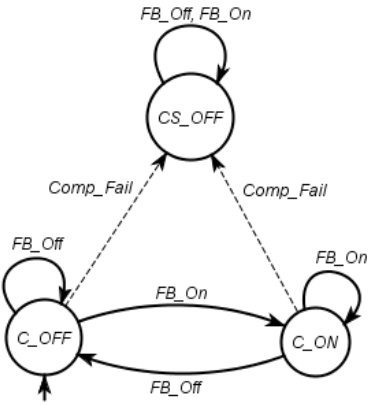
Fonte: Autor

queima na bobina do motor ou de algum dispositivo interno do compressor. O autômato rotulador para este caso é mostrado na Figura 4.26.

Para que seja possível distinguir a falha do compressor da falha do relé que o aciona, é necessário considerar também o modelo do relé como parte da planta, como mostrado na Figura 4.27.

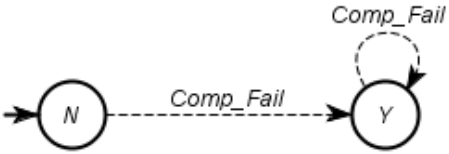
O diagnosticador obtido a partir dos modelos mostrados nas figuras 4.25 e 4.27, é composto somente por estados incertos (ver Figura 4.28), ou seja, a falha é não diagnosticável com base nos modelos propostos.

Figura 4.25: Autômato que Modela o Compressor, G_{Comp}



Fonte: Autor

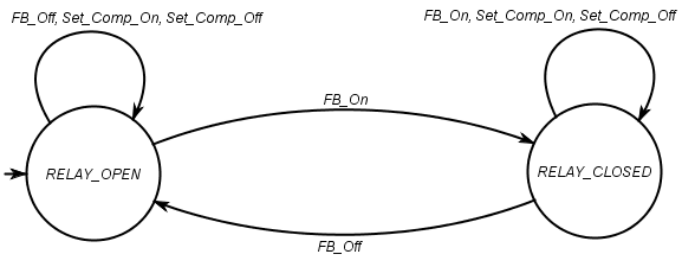
Figura 4.26: Autômato Rotulador referente a Falha do Compressor, $Al_{CompMotorModel}$



Fonte: Autor

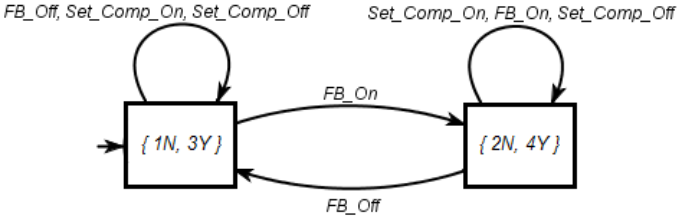
Para contornar este problema, é necessário aplicar o mapeamento de sensores novamente. Uma vez que o sistema possui dois sensores de temperatura, precisamos entender se eles podem prover algum tipo de informação adicional que seja relevante em termos de diagnóstico da falha do compressor e, caso positivo, qual a melhor maneira de se

Figura 4.27: Modelo para o Relé do Compressor, $G_{CompRelay}$



Fonte: Autor

Figura 4.28: Diagnosticador para o modelo $(G_{(Comp}||G_{CompRelay})$



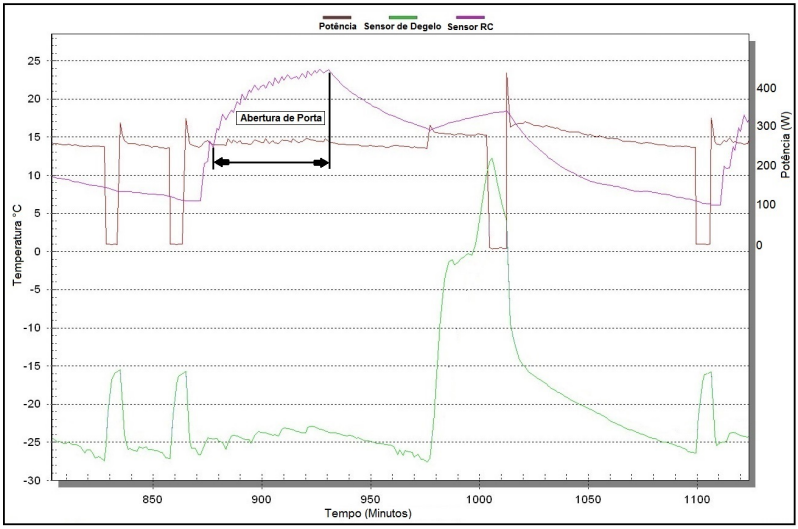
Fonte: Autor

utilizar essa informação no modelo do diagnosticador.

Na Figura 4.29 é mostrado o comportamento da temperatura em diferentes situações de uso do refrigerador. Neste gráfico, vemos o comportamento da temperatura lida pelo *Defrost Sensor* (Sensor de Degelo), que está localizado junto ao evaporador do sistema de refrigeração e também o comportamento da temperatura lida pelo *RC Sensor* (Sensor do Compartimento Refrigerador) localizado dentro do compartimento refrigerador. Como se pode notar, conforme o compressor liga e desliga, o

Defrost Sensor possui uma resposta mais rápida às mudanças de temperatura, em relação ao *RC Sensor*. Assim, por se mostrar mais sensível a estas variações de temperatura, o sensor de degelo se mostra mais adequado para o mapeamento.

Figura 4.29: Comportamento das Temperaturas do Refrigerador

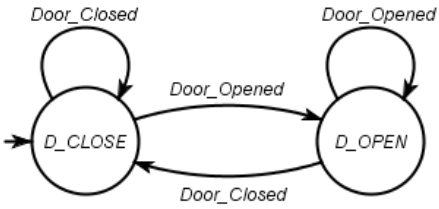


Fonte: Autor

Ainda, neste gráfico pode-se observar o perfil da temperatura em relação à abertura de porta. Mesmo estando o compressor ligado, constantes aberturas de porta fazem a temperatura aumentar. Como o refrigerador é provido de circuito de leitura do estado da porta (via *feedback* do interruptor de porta), podemos considerar o estado da porta no mapeamento a fim de evitar um erro de interpretação entre a temperatura lida e o real estado do compressor. O autômato que modela o comportamento da porta é mostrado na Figura 4.30.

Finalmente, o modelo completo para o compressor G_C mostrado

Figura 4.30: Autômato que Modela o Comportamento da Porta, G_{Door}



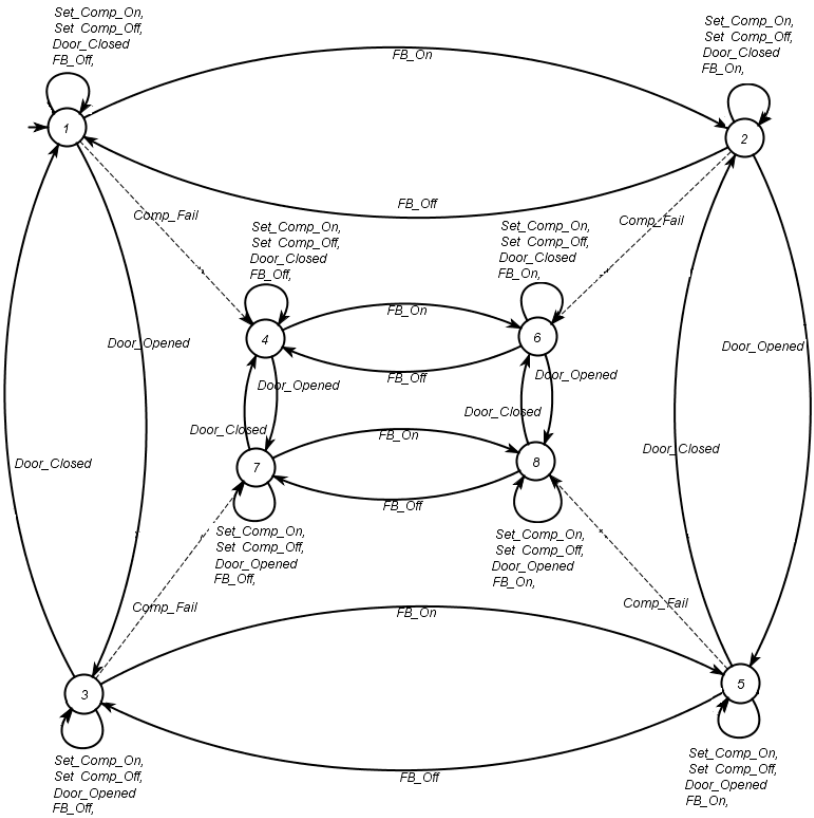
Fonte: Autor

na Figura 4.31, é obtido compondo-se os modelos do compressor, do relé e da porta, ou seja, $G_C = G_{Comp}||G_{CompRelay}||G_{Door}$.

A Tabela 4.4 mostra o mapeamento de sensor considerado no modelo de G_C, onde foram mapeados a variação de temperatura sob o sensor de degelo e o estado da porta. Nesta tabela, *Delta+* e *Delta-* representam uma variação positiva e negativa respectivamente sob o sensor de degelo. Os estados 4, 6, 7 e 8 referem-se aos estados onde há a falha do compressor, indicado por CS_OFF. Estes estados estão associados à ocorrência de uma variação positiva de temperatura $Delta+$, contudo esta variação pode ser também devido a outros fatores que não a falha do compressor propriamente dita. No caso do estado 4, esta variação poderia ser devido ao fato de que o relé responsável por ligar o compressor está desligado ($RELAY\_OPEN$). Já o estado 8, o aumento da temperatura poderia ser também devido a uma abertura de porta (D_OPEN). Por fim, o estado 6 representa uma condição onde o aumento de temperatura é somente devido à falha do compressor uma vez que o relé de acionamento do compressor está ligado ($RELAY\_CLOSED$) e a porta fechada (D_CLOSE).

Aplicando-se o mapeamento de sensores definido na Tabela 4.4 no modelo G_C e fazendo a composição síncrona deste com o autômato

Figura 4.31: Modelo Final para o Compressor, G_C



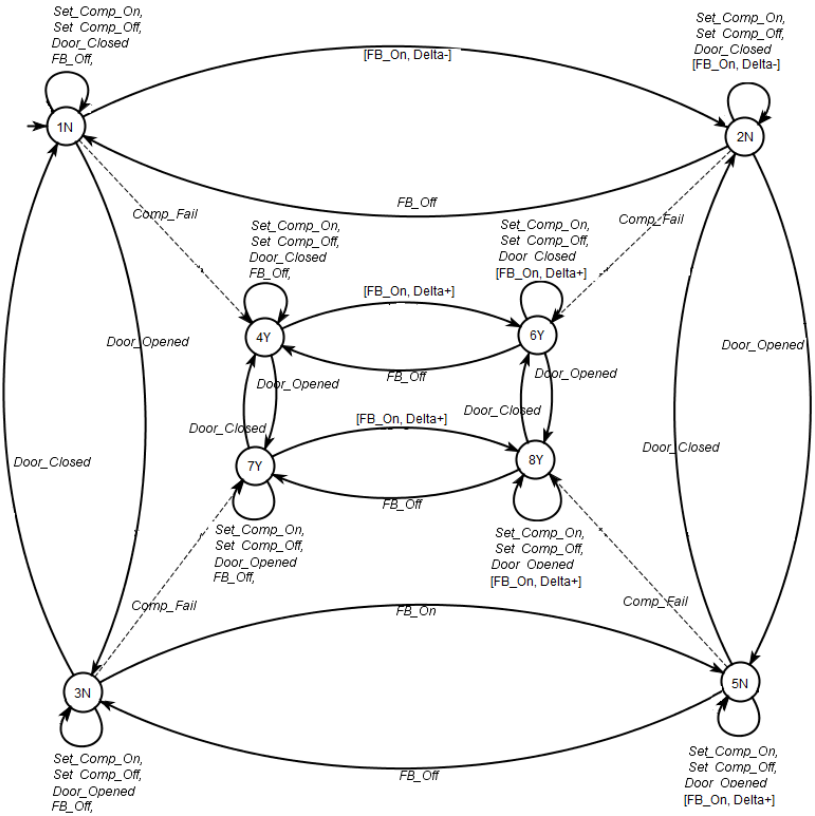
Fonte: Autor

rotulador $Al_{CompMotorModel}$, obtém-se o autômato G_C_Map, mostrado na Figura 4.32.

Por fim, o diagnosticador Gd_{Comp} para o compressor, mostrado na Figura 4.33, foi obtido calculando-se o observador de G_C_Map.

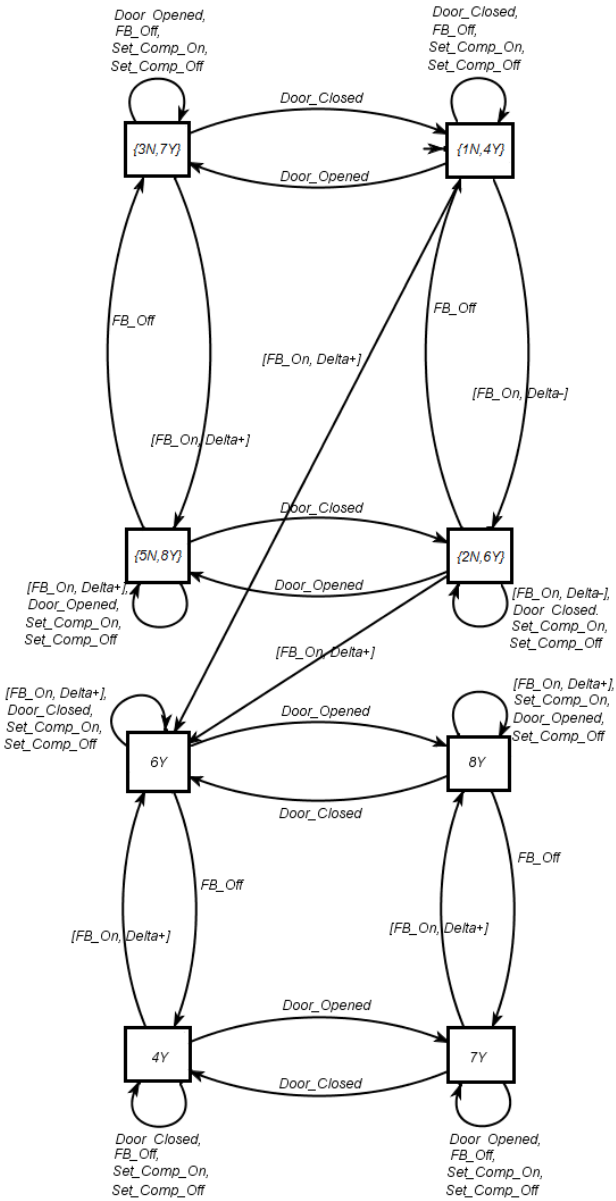
É possível ainda simplificar o diagnosticador obtido agrupando-

Figura 4.32: Autômato G_C_Map resultante de $G_C||Al_{CompMotorModel}$



Fonte: Autor

Figura 4.33: Diagnosticador para o Compressor, Gd_{Comp}



Fonte: Autor

Tabela 4.4: Mapeamento de Sensor para o Modelo do Compressor

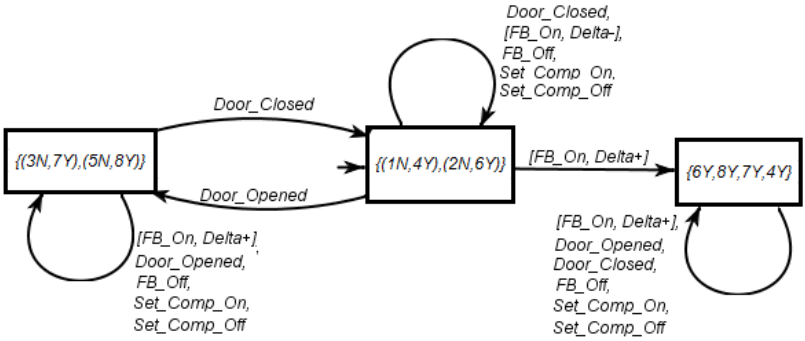
Estado	Mapeamento
$1-(C\_OFF,RELAY\_OPEN,D\_CLOSE)$	$[FB\_Off, Delta+]$
$2-(C\_ON,RELAY\_CLOSED,D\_CLOSE$	$[FB\_On, Delta-]$
$3-(C\_OFF,RELAY\_OPEN,D\_OPEN)$	$[FB\_Off, Delta+]$
$4-(CS\_OFF,RELAY\_OPEN,D\_CLOSE)$	$[FB\_Off, Delta+]$
$5-(C\_ON,RELAY\_CLOSED,D\_OPEN)$	$[FB\_On, Delta+]$
$6-(CS\_OFF,RELAY\_CLOSED,D\_CLOSE)$	$[FB\_On, Delta+]$
$7-(CS\_OFF,RELAY\_OPEN,D\_OPEN)$	$[FB\_Off, Delta+]$
$8-(CS\_OFF,RELAY\_CLOSED,D\_OPEN)$	$[FB\_On, Delta+]$

se os estados incertos {3N,7Y} e {5N, 8Y} e também {1N,4Y} e {2N,6Y}. Analogamente para os estados certos 6Y, 8Y , 4Y e 7Y, como mostrado na Figura 4.34. Contudo, os auto-laços dos estados {(3N,7Y), (5N, 8Y)}, {(1N,4Y), (2N,6Y)} e {6Y,8Y,4Y,7Y} não possuem nenhum efeito do ponto de vista de diagnóstico e podem no entanto ser omitidos do modelo do diagnosticador deixando-o ainda mais simples, como mostrado na Figura 4.35. Essa simplificação é importante e deve ser realizada sempre que possível pois ajudará a economizar espaço de memória no microcontrolador na fase de implementação, conforme será discutido no Capítulo 5.

4.5.2 Diagnosticador da Resistência de Degelo

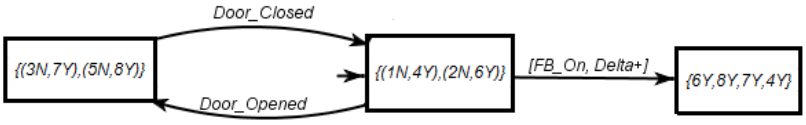
O autômato que modela a resistência de degelo, mostrado na Figura 4.36, é composto pelos estados *H_ON* (Resistência Ligada), *H_OFF* (Resistência Desligada) e *H_FAIL* (Resistência em Falha). Note que o estado *H_FAIL* considera apenas a situação onde a resistência não ligou quando era esperado que a mesma ligasse. Quando o diagnosticador da resistência detecta uma falha, implica que a falha está realmente na resistência e não em algum dos elementos das camadas de *hardware* (relé) ou *software* (rotina de degelo). Em outras palavras, caso a resistência não esteja fisicamente danificada, mas não é observado seu acionamento quando solicitado, isto implica que há uma falha ou no circuito de *hardware* responsável pelo seu acionamento (relé) ou ainda uma falha

Figura 4.34: Diagnosticador do Compressor Minimizado, $Gd_{CompMin}$



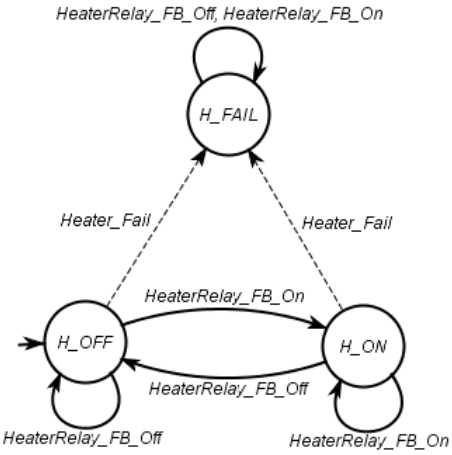
Fonte: Autor

Figura 4.35: Diagnosticador do Compressor Minimizado e Simplificado, $Gd_{CompMinSimple}$



Fonte: Autor

Figura 4.36: Autômato que modela a Resistência de Degelo, G_{Heater}



Fonte: Autor

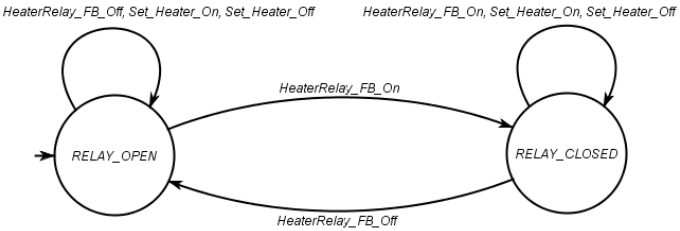
de *software* (rotina de degelo) e, portanto, os diagnosticadores responsáveis por estas falhas deverão identificá-la. Agora, para o caso onde haja realmente uma falha na resistência que faça com a mesma permaneça constantemente desligada mesmo havendo tensão sobre seus terminais, implica que o diagnosticador da resistência deve detectar essa falha.

Para que seja possível distinguir a falha da resistência da falha do relé que a aciona, é necessário considerar também o modelo do relé que aciona a resistência, como mostrado da Figura 4.37.

A composição síncrona entre o modelo da resistência e de seu respectivo relé pode ser visto na Figura 4.38.

Este modelo também requer um mapeamento de sensor, conforme mostrado na Tabela 4.5. Este mapeamento foi obtido criando-se um sensor virtual para gerar os eventos referentes à rotina de degelo e também usando a informação do sensor de degelo. Este último foi es-

Figura 4.37: Autômato que Modela o Relé da Resistência de Degelo, $G_{HeaterRelay}$

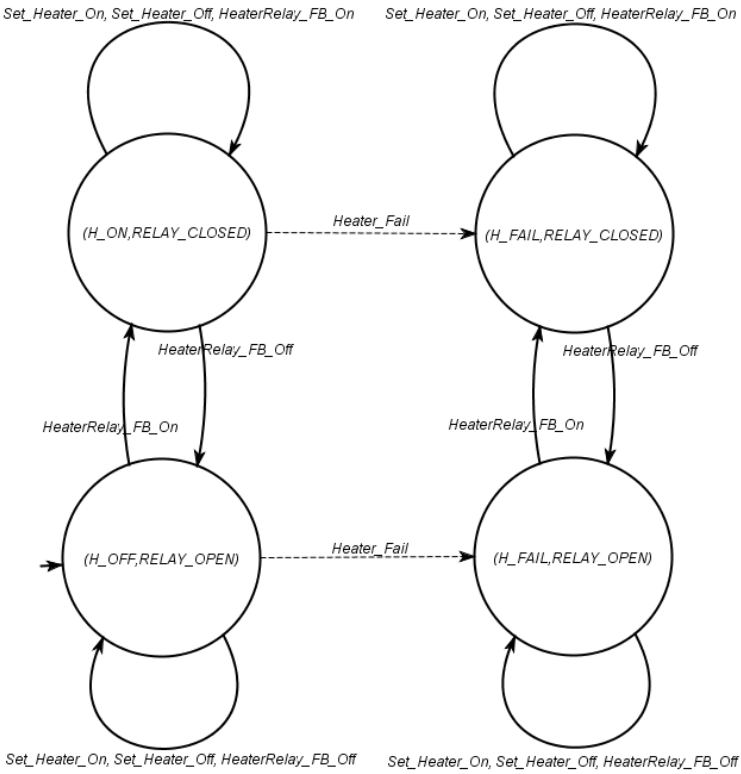


Fonte: Autor

colhido por duas razões: primeiro por ser o sensor que está em contado com o evaporador, que é o mesmo lugar onde a resistência de degelo está inserida, então a variação de temperatura neste sensor é maior quando comparada com o outro sensor (*RC Sensor*); e segundo por ser este o sensor que lê a temperatura de fim de degelo. Na rotina de degelo foram mapeados os eventos de início de degelo (Def_Start), de degelo finalizado por temperatura ($Def\_End\_By\_Temp$) e de degelo finalizado por tempo limite ($Def\_End\_By\_Timeout$). Já em relação ao sensor de degelo, sua leitura de temperatura é comparada com um valor de referência T_{ref}. Esta referência se refere à máxima temperatura que pode ser alcançada durante um degelo quando a resistência não está aquecendo (não está, portanto, ligada).

O tempo máximo de degelo (*DEF_TIMEOUT*) é um parâmetro que depende da plataforma do produto, para o refrigerador considerado neste estudo de caso, este valor é definido como 50 minutos. Para definir o valor de T_{ref}, foi monitorado o comportamento real do refrigerador durante consecutivos ciclos de degelo, quando a resistência foi mantida desligada e portanto todos os degelos tiveram duração de 50 minutos. Como se pode observar no gráfico da Figura 4.39, o sensor de degelo não registrou temperaturas maiores que -7°C em nenhum dos casos,

Figura 4.38: Autômato G_H obtido pela composição $G_{Heater}||G_{HeaterRelay}$

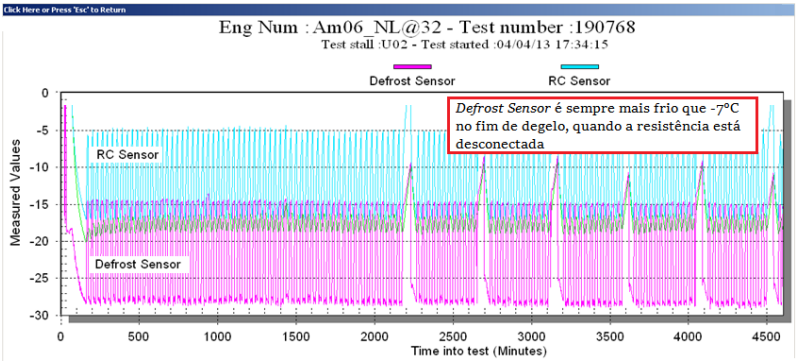


Fonte: Autor

Tabela 4.5: Mapeamento de Sensor para Resistência de Degelo

Estado	Mapeamento	Evento
$(H\_OFF, RELAY\_OPEN)$	$HeaterRelay\_FB\_Off$ AND $[Def\_End\_By\_Temp$ OR $Def\_End\_By\_Timeout]$	C1
$(H\_ON, RELAY\_CLOSED)$	$HeaterRelay\_FB\_On$ AND Def_Start	C2
$(H\_FAIL, RELAY\_OPEN)$	$HeaterRelay\_FB\_Off$ AND $Def\_End\_By\_Timeout$ AND $Def\_End\_Temp < T_{ref}$	C3
$(H\_FAIL, RELAY\_CLOSED)$	$HeaterRelay\_FB\_On$ AND Def_Start	C2

Figura 4.39: Comportamento do Degelo para o Caso de Resistência Desconectada

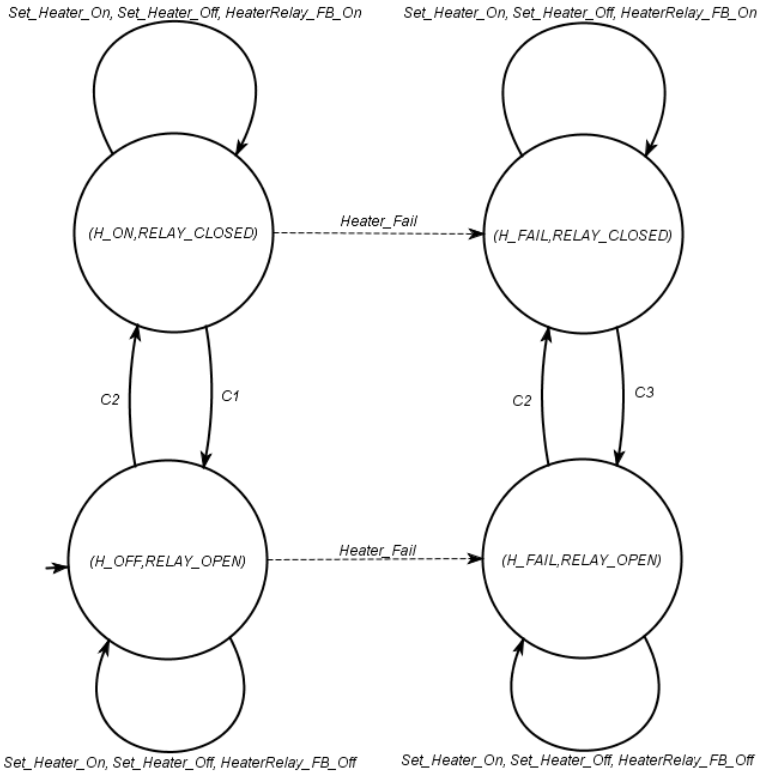


Fonte: Autor

assim definiu-se este valor como T_{ref}.

Na Figura 4.40 é mostrado o modelo final da resistência já considerando o mapeamento de sensor.

Figura 4.40: Autômato G_{H_Map} que modela a Resistência de Degelo considerando Mapeamento de Sensor

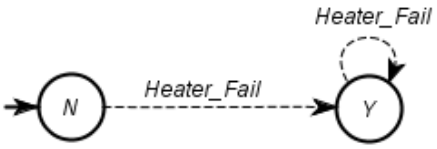


Fonte: Autor

O autômato rotulador é mostrado na Figura 4.41 e o autômato obtido pela composição síncrona $G_{H\_Map}||Al_{Heater}$ é mostrado na Figura 4.42. Finalmente o diagnosticador Gd_Heater da Figura 4.43 é obtido calculando-se $Obs(G_{H\_Map}||Al_{Heater})$.

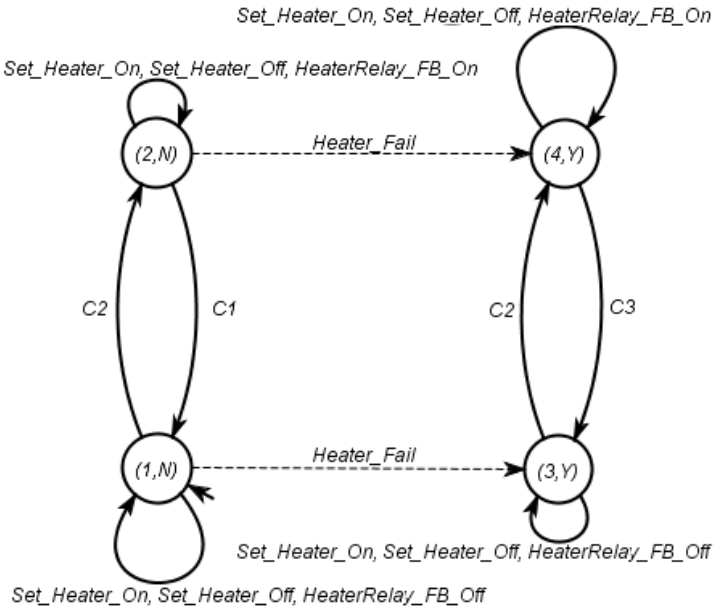
Entretanto, os auto-laços nos estados {1N,3Y},{2N,4Y}, {3Y} e {4Y}, não possuem nenhum efeito no ponto de vista de diagnóstico da resistência de degelo e podem portanto ser removidos. Além disso, os estados {3Y} e {4Y} podem ser agrupados num estado único {3Y,4Y}, simplificando assim o modelo do diagnosticador. A Figura 4.44 mostra o diagnosticador final considerando estas simplificações.

Figura 4.41: Autômato Rotulador para Falha da Resistência de Degelo, Al_{Heater}



Fonte: Autor

Figura 4.42: Autômato obtido pela composição $G_{H\_Map}||Al_{Heater}$

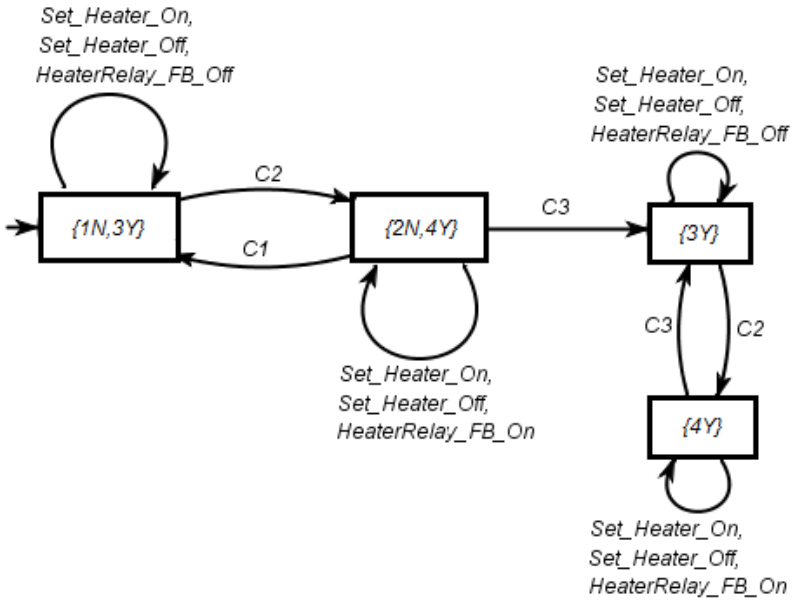


Fonte: Autor

4.6 Conclusões do Capítulo

Neste capítulo foi feita uma breve revisão dos conceitos básicos relativos ao funcionamento de um refrigerador *Frost Free*, mostrado seus principais componentes e a função que cada um desempenha do sistema. Foi detalhado também o comportamento das principais rotinas de controle de um refrigerador: termostática e degelo. Em seguida, foi aplicado o sistema de diagnóstico multicamadas apresentado do Capítulo 3 para o refrigerador proposto neste estudo de caso. Por fim, foi detalhado o projeto de cada diagnosticador, segundo a teoria de diagnose de falhas em SEDs proposto por Sampath et al. (1996). Viu-se também que em alguns casos, a fim de se obter um modelo de planta que seja mais ade-

Figura 4.43: Diagnosticador da Resistência de Degelo, Gd_{Heater}



Fonte: Autor

Figura 4.44: Diagnosticador Simplificado da Resistência de Degelo, $Gd_{HeaterSimple}$

Fonte: Autor

quado para a diagnose de falhas, é necessário obter um modelo com o mapeamento de sensores. Com a obtenção de todos os diagnosticadores propostos, verificou-se que não só é possível dividir o sistema em camadas e projetar diagnosticadores individuais para os elementos de cada camada como também observou-se que o método proposto por Sampath et al. (1996) também é aplicável mesmo quando subdividimos o sistema (em camadas neste caso) e também quando se considera um maior grau de abstração do sistema, como é o caso dos diagnosticadores da camada de *software*, onde as informações das variáveis de *software* foram consideradas como eventos provenientes de sensores virtuais. O próximo capítulo será dedicado à apresentação de uma metodologia para a implementação do *software* que modela os diagnosticadores obtidos em forma de autômatos bem como uma ferramenta para a geração automática deste *software*. Finalmente será mostrado como esta modelagem se relaciona com a arquitetura de diagnóstico proposta no Capítulo 3.

5 METODOLOGIA DE IMPLEMENTAÇÃO DO SISTEMA DE DIAGNÓSTICO MULTICAMADAS

Uma vez projetados os diagnosticadores, o próximo passo consiste em traduzi-los para uma linguagem de *software* para então serem embarcados em um microcontrolador. Esta é uma tarefa crítica, pois é necessário garantir que esta tradução não inclua erros em relação aos modelos dos diagnosticadores. A fim de garantir um padrão na implementação deste *software* é necessário criar então um modelo que possa ser aplicado para qualquer diagnosticador concebido sob a metodologia de diagnóstico de falhas em SEDs. Este capítulo apresenta o modelo desenvolvido para este propósito e que pode ser aplicado à arquitetura multicamadas de diagnóstico mostrada no Capítulo 3.

5.1 Software de Implementação dos Diagnosticadores

Normalmente o *software* de um sistema embarcado é composto por um conjunto de pequenas rotinas de *software*, onde cada uma realiza uma tarefa específica e possui o mínimo de dependência possível com as demais. Estas pequenas rotinas, também conhecidas como módulos, podem ser agrupadas a fim de formarem bibliotecas e são desenvolvidas de modo que possam ser parametrizadas ou configuradas externamente, conforme as necessidades da aplicação na qual serão inseridas. Desta forma, evita-se que o código fonte da biblioteca seja indevidamente alterado, garantindo assim não só a integridade do *software* como também o reuso destas bibliotecas em outros projetos. Grande parte dos sistemas embarcados são desenvolvidos utilizando linguagem ANSI C (KERNIGHAN e RITCHIE, 1988) como padrão (BARR, 2006). Nesta linguagem é comum se estruturar uma biblioteca com os seguintes arquivos:

• *lib_name*.c e *lib_name*.h: arquivos que implementam os algoritmos cujo conteúdo não pode ser alterado;

• *lib_name*_prv.h (cabeçalho de parâmetros privado) e *lib_name*_prm.h

(cabeçalho de parâmetros públicos): arquivos de cabeçalhos onde são inseridos os parâmetros e configurações. Esses arquivos podem ser modificados pelo implementador (usuário da biblioteca).

Retornando ao problema da implementação dos autômatos diagnosticadores em *software*, para que uma solução possa ser aplicável a qualquer sistema embarcado é necessário que se crie uma biblioteca específica que seja capaz de manipular qualquer autômato diagnosticador da mesma maneira, ou seja, um algoritmo único que gerencie e realize a evolução dos estados de cada autômato diagnosticador. Essa biblioteca deve permitir também que as informações referentes à cada autômato (nome dos estados, nome dos eventos, função de transição, relação de eventos ativos, etc) possam ser configuráveis externamente segundo um padrão, sem que seja necessário realizar alterações no algoritmo principal. É importante também que essa biblioteca não tenha dependência com o tipo do microcontrolador no qual ela será embarcada, em outras palavras, a biblioteca não deve acessar diretamente nenhum registrador ou periférico do microcontrolador.

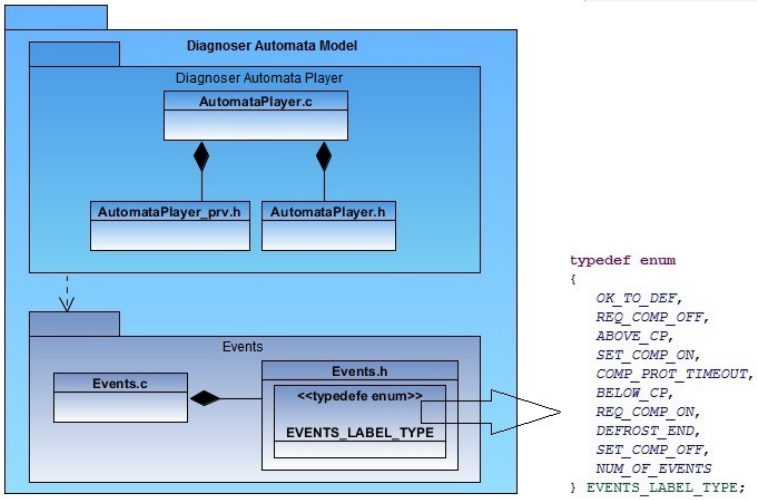
Para atender os requisitos acima descritos, é proposto neste trabalho um modelo para *software* em linguagem ANSI C, denominado DAM (*Diagnoser Automata Model*), cujo diagrama de classe está representado na Figura 5.1. O DAM é composto pelas bibliotecas *Diagnoser Automata Player* e *Events*.

A biblioteca *Diagnoser Automata Player* por sua vez é composto pelos seguintes arquivos:

• *AutomataPlayer.c*: Implementa o algoritmo que trata a evolução dos estados dos autômatos diagnosticadores, conforme a ocorrência dos eventos, ou seja, lê os eventos registrados pelo módulo *Events* e atualiza adequadamente o estado de cada autômato diagnosticador. Este módulo também implementa o método (função) para informar quando um autômato diagnosticador chegou a um estado que é certo da ocorrência da falha;

• *AutomataPlayer.h*: Contém os protótipos das funções públicas imple-

Figura 5.1: Diagrama de Classe do DAM



Fonte: Autor

mentadas no *AutomataPlayer.c* para fornecer a API *(Application Programming Interface*) para os demais módulos do *software*;

• *AutomataPlayer_prv.h*: Onde todas as tabelas de estados e transições para todos os diagnosticadores são implementadas seguindo um formato de representação padrão.

Já a biblioteca *Events* refere-se ao bloco *Manipulador de Eventos* mostrado na Figura 3.3 e é composto por:

• *Events.c*: Implementa as funções (algoritmos) para ler e adicionar no *buffer* os eventos do sistema necessários para o diagnóstico. Também implementa as funções relativas ao mapeamento de sensores, quando necessário;

• *Events.h*: Contém a lista de eventos necessária para o diagnóstico e os protótipos (API) das funções usadas para escrever e ler os eventos no *buffer*.

A função *"void AutomataPlayer__Handler(void)"*, mostrada na Figura 5.2, implementa o algoritmo de evolução dos autômatos diagnosticadores. Esta função constantemente lê os eventos do *buffer* e atualiza os estados de todos os autômatos com base nas tabelas definidas em *AutomataPlayer_prv.h*. É importante ressaltar que esta função é única e capaz de manipular até 256 autômatos por vez, ou seja, é necessário implementar (instalar o módulo *AutomataPlayer.c*) apenas uma vez, já que os diagnosticadores estão mapeados em forma de tabela no arquivo de cabeçalho *AutomataPlayer_prv.h*. A limitação será dada então pelo espaço de memória e capacidade de processamento do microcontrolador usado.

Figura 5.2: Rotina que Manipula os Autômatos Diagnosticadores

```
void AutomataPlayer__Handler(void)
{
 unsigned short int event;
 unsigned short int new_state;
 unsigned char event_count;
 unsigned char automata_count;

 event = Events__GetEvent();
 while(event != EVENTS_NO_EVENT)
 {
  //Events
  for(automata_count = 0;automata_count < AUTOMATA_PLAYER_NUMBER_OF_AUTOMATAS;automata_count++)
  {
   event_count = 0;
   while(Automatas[automata_count][Automata_State[automata_count]][event_count].Event != EVENTS_NO_EVENT)
   {
    if(Automatas[automata_count][Automata_State[automata_count]][event_count].Event == event)
    {
     new_state = Automatas[automata_count][Automata_State[automata_count]][event_count].TargetState;
     if(new_state != Automata_State[automata_count])
     {
      Automata_State[automata_count] = new_state;
      PUBLISH_NEW_STATE(automata_count);
      break; //End this automata search
     }
    }
     event_count++;
   }
  }
    event = Events__GetEvent();
 }
}
```

Fonte: Autor

O algoritmo mostrado na Figura 5.2 utiliza a técnica de ponteiro de função, onde para cada estado há uma lista de transições que contém o evento e o estado de destino. Para cada lista há uma transição de fechamento usada para concluir a busca, para esta transição de fechamento é usado um evento desconhecido chamado **EVENT_NO_EVENT** (MAAS et al., 2012). Esta lista é implementada no arquivo *AutomataPlayer_prv.h* e um exemplo de sua estrutura é mostrado na Figura 5.3. Como se pode observar, cada estado do autômato é mapeado considerando o conjunto de eventos associados com o estado atual e o estado destino após a ocorrência de um desses eventos.

Figura 5.3: Formato da Lista de Transições Relativa aos Eventos

```
const AUTOMATA_PLAYER_TRANSITION_TYPE Automata_1_State_0_Event_List[] =
{
    {OK_TO_DEF, 1},
    {ABOVE_CP, 2},
    {REQ_COMP_ON, 3},
    {BELOW_CP, 0},
    {DEFROST_END, 4},
    {SET_COMP_OFF, 0},
    {EVENTS_NO_EVENT, 0}
};

const AUTOMATA_PLAYER_TRANSITION_TYPE Automata_1_State_1_Event_List[] =
{
    {OK_TO_DEF, 4},
    {ABOVE_CP, 1},
    {REQ_COMP_ON, 3},
    {BELOW_CP, 1},
    {DEFROST_END, 2},
    {SET_COMP_OFF, 1},
    {EVENTS_NO_EVENT, 0}
};
```

Fonte: Autor

Para exemplificar como esta lista funciona, considere a lista Automata_1_State_0_Event_List[], mostrada na Figura 5.3, que mapeia todos os eventos referentes ao estado 0 do autômato 1, como: *OK_TO_DEF*, *ABOVE_CP*, *REQ_COMP_ON*, etc. O número que aparece ao lado de cada evento, identifica o estado de destino para o qual o autômato deve

mudar quando ocorrer o evento em questão. Por exemplo, a partir do atual estado 0, se o evento *OK_TO_DEF* ocorrer, então o autômato mudará para o estado 1.

O algoritmo que manipula as transições dos autômatos diagnosticadores usa uma lista, também implementada em *AutomataPlayer_prv.h*, que aponta para um dado estado, como mostrado na Figura 5.4.

Figura 5.4: Lista de Estados Relativos a cada Autômato

```
//  Compressor
AUTOMATA_PLAYER_TRANSITION_TYPE * const Automata_1_States[] =
{
    (AUTOMATA_PLAYER_TRANSITION_TYPE *const) Automata_1_State_0_Event_List,
    (AUTOMATA_PLAYER_TRANSITION_TYPE *const) Automata_1_State_1_Event_List,
    (AUTOMATA_PLAYER_TRANSITION_TYPE *const) Automata_1_State_2_Event_List
};

//  Heater
AUTOMATA_PLAYER_TRANSITION_TYPE * const Automata_2_States[] =
{
    (AUTOMATA_PLAYER_TRANSITION_TYPE *const) Automata_2_State_0_Event_List,
    (AUTOMATA_PLAYER_TRANSITION_TYPE *const) Automata_2_State_1_Event_List,
    (AUTOMATA_PLAYER_TRANSITION_TYPE *const) Automata_2_State_2_Event_List
};
```

Fonte: Autor

Assim, supondo que o autômato 1 esteja no estado 0 (isto é *Automata_1_States[0]*), ele estará apontando para a lista *Automata_1_State_0_Event_List*, que é mesma anteriormente mostrada na Figura 5.3. Quando um novo evento ocorre, o *software* faz uma busca na lista de eventos deste estado, caso este evento esteja presente nesta lista, o estado atual torna-se então o estado de destino, conforme definido na lista de transição.

Na Figura 5.5 é mostrado um exemplo da lista de eventos necessários para o diagnóstico que deve ser implementada no arquivo *Events.h*. Esta consiste em uma lista enumerada com definição de tipo (*typedef*

enum), cujo tipo é *EVENTS_LABEL_DEF*.

Figura 5.5: Exemplo de Implementação da Lista de Eventos

```
typedef enum
{
    FB_ON_AND_POSITIVE_DELTA = 0,
    FB_DOOR_OPENED,
    FB_DOOR_CLOSED,
    DEF_END_BY_TIMEOUT,
    DEF_END_BY_TEMP,
    HEATER_RELAY_FB_ON_AND_DEF_START,
    HEATER_RELAY_FB_OFF_AND_DEF_END_BY_TIMEOUT,
    HEATER_RELAY_FB_OFF_AND_DEF_END_BY_TEMP,
    C1,
    C2,
    C3,
    OK_TO_DEF,
    DEFROSTING_COMP_ON,
    BELOW_CP_COMP_ON,
    ABOVE_CP_COMP_ON,
    DEFROSTING_COMP_OFF,
    ABOVE_CP,
    BELOW_CP,
    DEFROST_END,
    ABOVE_CP_COMP_OFF,
    BELOW_CP_COMP_OFF,
} EVENTS_LABEL_DEF;
```

Fonte: Autor

Conforme mencionado anteriormente, o *Events.c* implementa os algoritmos para remover e adicionar eventos em um *buffer*, definidos respectivamente pelas funções *"unsigned short int Events__GetEvent(void)"* e *"void Events__Queue(unsigned short int event)"*.

Neste módulo devem ser implementados também os algoritmos para ler os eventos necessários do sistema e convertê-los (caso preciso) no formato segundo o padrão estabelecido na lista de eventos definida no arquivo *Events.h*. Na Figura 5.6 mostra-se parte do *software* que implementa o algoritmo usado para gerar os eventos para o diagnosticador do

Heater, conforme o mapeamento de sensor da Tabela 4.5. Note que, uma vez que o evento é detectado, a função *Events__Queue* é usada para adicioná-lo ao *buffer*.

Para que seja possível externar o estado atual de cada diagnosticador, bem como se algum deles detectou uma falha, há duas funções implementadas no módulo *AutomataPlayer.c*, mostradas na Figura 5.7. A primeira função retorna o estado atual para o autômato diagnosticador informado no parâmetro *automata*. Já a segunda função irá retornar Verdadeiro (*TRUE*) caso o estado do autômato diagnosticador informado no parâmetro *automata* for um estado certo de falha, caso contrário, retornará Falso (*FALSE*).

5.2 Ferramenta de Geração Automática de Código

Conforme mostrado anteriormente, a configuração da biblioteca *Diagnoser Automata Player* é feita através do arquivo *AutomataPlayer_prv.h*. Este é um processo crítico, uma vez que é neste arquivo que estarão descritas todas as tabelas de estados e transições dos autômatos diagnosticadores e o menor erro no preenchimento deste arquivo pode comprometer o diagnóstico. Outro arquivo que precisa ser manualmente preenchido é o *Events.h*, nele o usuário deve implementar a lista de eventos necessária para o diagnóstico. Assim, é mais seguro e produtivo ter uma ferramenta que seja capaz de gerar automaticamente os conteúdos destes arquivos a partir dos modelos dos diagnosticadores, tornando o processo mais robusto. Para este propósito, foi desenvolvida neste trabalho uma ferramenta capaz de gerar automaticamente esses arquivos.

Esta ferramenta, chamada de *Doctor Who*, mostrada na Figura 5.8, tem como entrada os autômatos diagnosticadores e como saída os arquivos cabeçalho *AutomataPlayer_prv.h* e *Events.h*. Atualmente os arquivos referentes aos autômatos diagnosticadores devem estar no formato .xmd do programa IDES (2010), usado para projetar o autômato diagnosticador. É importante ressaltar neste momento que, para o correto funcionamento da ferramenta proposta, é necessário que todos os

Figura 5.6: Parte do *Software* do Módulo *Events* Relativo à Implementação do Modelo do *Heater*

```
switch(Heater_Relay_State)
{
case RELAY_WAIT_FB:
    if(Lsi__GetDigitalInput(HEATER_RELAY_FDBK) == RELAY_FEEDBACK_OFF)
    {
        Heater_Relay_State = RELAY_OPEN;
    }
    else if(Lsi__GetDigitalInput(HEATER_RELAY_FDBK) == RELAY_FEEDBACK_ON)
    {
        Heater_Relay_State = RELAY_CLOSED;
    }
    break;

case    RELAY_CLOSED: // Heater_Relay_Feedback_On
    if((Defrost__GetDefrostState() == DEFROST_STATE_DEFROSTING)&&
       (Def_Status_Updated == FALSE))
    {
        Def_Status_Updated = TRUE;

        //Heater_Relay_FB_On AND Def_Start
        Events__Queue(C2);
    }
    break;

case    RELAY_OPEN: //Heater_Relay_Feedback_Off
    if((Defrost__GetDefrostState() == DEFROST_STATE_DROPPING)&&
       (Def_Status_Updated == FALSE))
    {
        Def_Status_Updated = TRUE;
        if(Defrost__GetDefrostDuration() >= DEF_TIMEOUT)//Def End by Timeout
        {
            if(Sensor__GetValue(DEF_SENSOR) < (T_REF))//Def_End_Temp < T_ref
            {
      //Heater_Relay_FB_Off AND Def_End_By_Timeout AND Def_End_Temp < T_ref
                Events__Queue(C3);
            }
            else
            {
                //Heater_Relay_FB_Off AND Def_End_By_Timeout
                Events__Queue(C1);
            }
        }
        else    //Def End by Temp
        {
            //Heater_Relay_FB_Off AND Def_End_By_Temp
            Events__Queue(C1);
        }
    }
    break;

default:
    break;
}
```

Fonte: Autor

Figura 5.7: Funções que retornam os estado de um dado autômato e resultado do diagnóstico

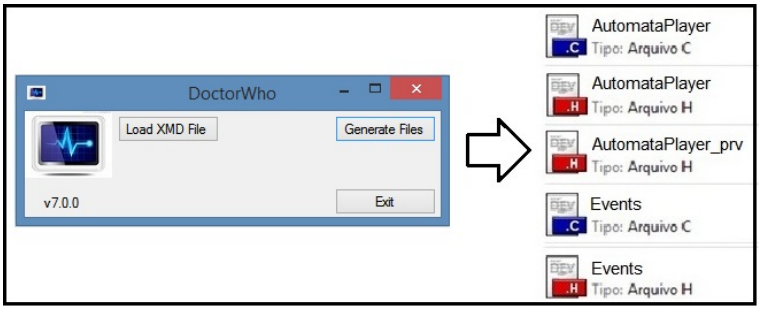
```
unsigned short int AutomataPlayer__GetAutomataState(unsigned char automata)
{
    return Automata_State[automata];
}


unsigned char AutomataPlayer__IsFailState(unsigned char automata)
{
    unsigned char ret_value;
    unsigned char state;
    unsigned char actual_state;

    ret_value = FALSE;
#if(MAX_NUM_OF_FAIL_STATE > 0)
    actual_state = Automata_State[automata];

    for(state = 0; state < MAX_NUM_OF_FAIL_STATE; state++)
    {
        if(Fail_State_List_Table[automata][state] == actual_state)
        {
            ret_value = TRUE;
            break;
        }
    }
#endif
    return ret_value;
}
```

Fonte: Autor

estados certos (da ocorrência da falha) do autômato diagnosticador sejam representados como estados marcados, isto é, assim que obtido o diagnosticador G_d, os estados que possuem apenas rótulos *Y*, devem ser configurados no IDES como estados marcados.

Esta ferramenta também gera os arquivos *AutomataPlayer.c* e *AutomataPlayer.h*, porém esses arquivos não devem ser modificados pelo usuário pois possuem o código fonte do algorítimo que manipula os autômatos diagnosticadores, conforme mencionado anteriormente. O arquivo *Events.c* também é gerado já com as funções *"Events__Queue"* e

Figura 5.8: Ferramenta de Geração Automática de Código



Fonte: Autor

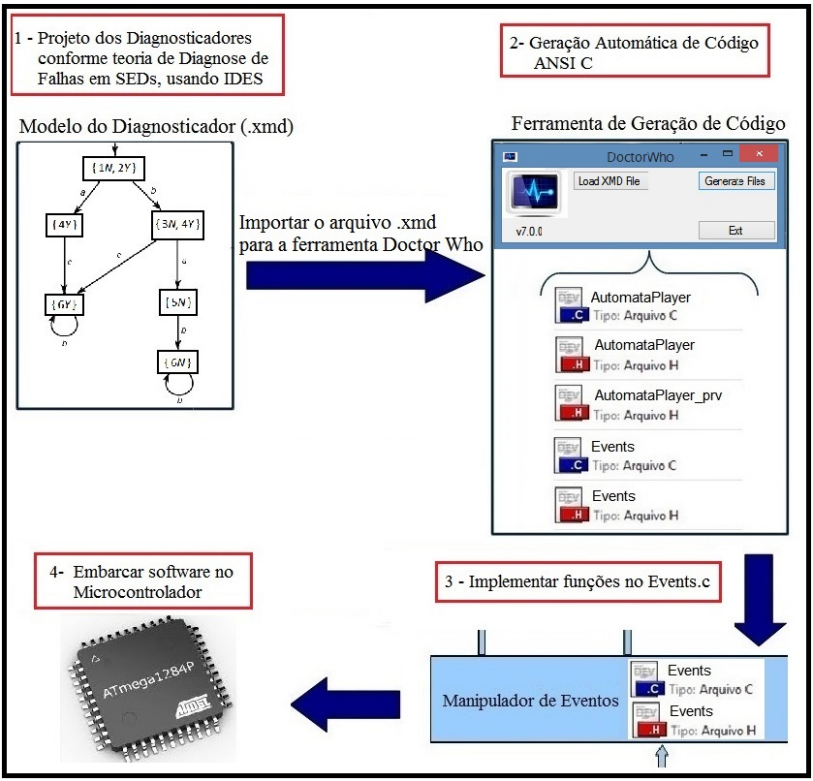
"Events__GetEvent" devidamente implementadas. O usuário será responsável apenas por implementar neste arquivo os métodos necessários para obter os eventos do sistema e convertê-los (se necessário) no formato de acordo com o padrão definido em *Events.h* e adicioná-los no *buffer*, conforme mencionado anteriormente.

Assim, obtemos uma solução completa para o projeto e implementação dos diagnosticadores para sistemas embarcados. Os diagnosticadores podem ser projetados através da teoria de diagnose de falhas em SEDs, tal como apresentado no Capítulo 2 e modelados com auxílio da ferramenta IDES. Por fim, importando-se os arquivos .xmd gerados pelo IDES na ferramenta *Doctor Who* gera-se o *software* em linguagem ANSI C que permite a implementação dos diagnosticadores junto ao sistema embarcado. O procedimento a ser seguido é ilustrado na Figura 5.9.

5.3 Aplicação da Metodologia no Estudo de Caso

Uma vez definidas uma metodologia e uma estrutura para implementação em *software* tanto dos autômatos diagnosticadores como do Manipulador de Eventos e, ainda, em posse de uma ferramenta que

Figura 5.9: Sequência de Projeto e Implementação de Diagnosticadores

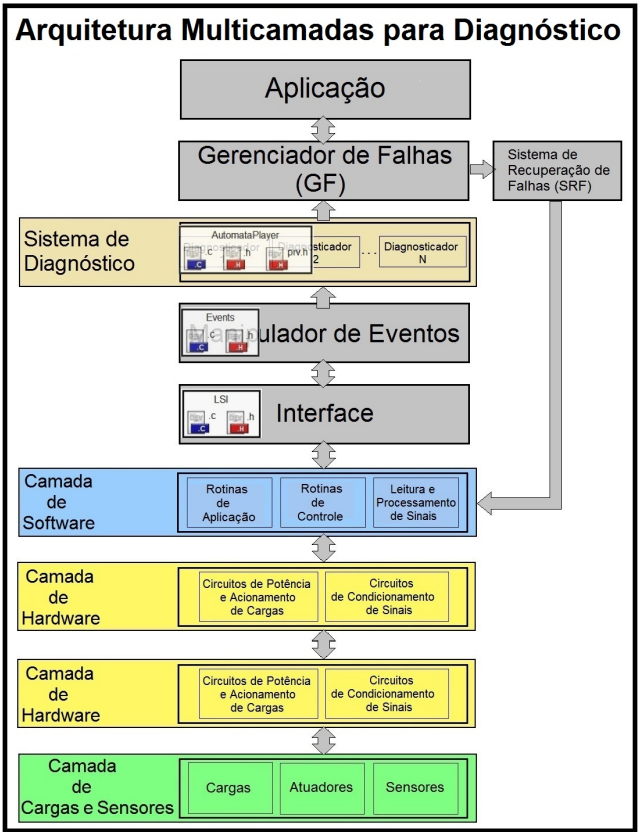


Fonte: Autor

permite gerar automaticamente estes códigos, pode-se aplicá-la para gerar os códigos de cada autômato diagnosticador projetado no Capítulo 4, para finalmente embarcá-lo no microcontrolador da placa de controle do refrigerador, juntamente com o código principal do produto. Seguindo a metodologia mostrada na Figura 5.9, o arquivo .xmd de cada diagnosticador foi importado para ferramenta *Doctor Who* e os arquivos *AutomataPlayer.c*, *AutomataPlayer.h*, *AutomataPlayer_prv.h*, *Events.c* e *Events.h* foram gerados. Em seguida, foram implementados no arquivo *Events.c* os métodos para obter e adicionar no *buffer* os eventos necessários do sistema, conforme listado no arquivo *Events.h*. Na Figura 5.10 é mostrada a arquitetura de diagnóstico completa aplicada ao estudo de caso proposto neste trabalho. Na camada de cargas têm-se o compressor e a resistência de degelo, já na camada de *hardware* os circuitos de acionamentos destas cargas (relés e circuitos de potência) e finalmente na camada de *software* as rotinas termostática, degelo e de controle do compressor. O bloco *Interface* neste caso é composto por um módulo de *software* denominado LSI (*Load Sensing Interface*), que possui as rotinas para realizar o acionamento das cargas e também a leitura dos sensores disponíveis na placa eletrônica. Por se tratar de uma biblioteca proprietária da Whirlpool, por questões de sigilo seu conteúdo não poderá ser mostrado neste trabalho, contudo tal omissão não representa ônus para o entendimento ou validação desta aplicação. Em seguida tem-se o bloco Manipulador de Eventos, onde é implementado o módulo *Events* que por sua vez se comunica com a camada de sistema de diagnóstico, onde estão implementados todos diagnosticadores, através da biblioteca *AutmataPlayer*.

O bloco Gerenciador de Falhas (GF) usado nesta aplicação, também de propriedade da Whirlpool, é responsável por monitorar o estado de cada diagnosticador e caso algum deles atinja um estado certo (da ocorrência da falha), irá associar a esta falha um código predefinido e armazená-lo em uma lista, segundo uma prioridade previamente estabelecida. Esta falha será então informada ao bloco SRF (Sistema de Recuperação de Falhas), que por sua vez tomará as ações necessárias para a recuperação do sistema, caso seja aplicável a esta falha.

Figura 5.10: Arquitetura de Diagnóstico Aplicada ao Estudo de Caso



Fonte: Autor

Figura 5.11: Exemplo de Externação de um Código de Falha

Fonte: Autor

A aplicação pode requisitar a qualquer momento ao GF as informações referentes ao diagnóstico. Para o produto utilizado no estudo de caso, os códigos de falhas armazenados no GF, podem ser mostrados para o usuário através de uma combinação de LEDs acesos e apagados no painel do produto, conforme mostrado na Figura 5.11, ou ainda extraídos através da comunicação serial, utilizando-se um dispositivo específico desenvolvido pela Whirlpool. Estes dados do diagnóstico permitem tanto que o técnico efetue uma manutenção corretiva mais assertiva como também fornecem informações à equipe de engenharia a fim de entender o modo de falha e corrigi-los nos projetos seguintes, tornando os produtos mais robustos.

Assim, temos finalmente a implementação completa da solução de diagnóstico de falhas multicamadas para sistemas embarcados, proposta neste trabalho.

5.4 Conclusões do Capítulo

Neste capítulo foi apresentada uma metodologia para traduzir em linguagem de *software* os diagnosticadores projetados segundo a metodologia de diagnose de falhas em SEDs apresentada no Capítulo 2. Com o intuito de obter um *software* de fácil integração com *software* principal do sistema embarcado e que fosse independente de microcontrolador e arquitetura usada neste sistema, foi criada uma biblioteca capaz de manipular os autômatos diagnosticadores e os eventos gerados pela planta. Para essa biblioteca, as tabelas de estado e transições dos autômatos diagnosticadores são lidas através de um arquivo de configuração privado *AutomataPlayer_prv.h*. Os arquivos desta biblioteca bem como o arquivo de configuração, podem ser automaticamente gerados através da ferramenta de geração automática de código *Doctor Who*. Em seguida esta metodologia foi aplicada ao estudo de caso mostrado no Capítulo 4. Embora a metodologia apresentada, bem como a geração automática de código foram desenvolvidas com foco em linguagem C, é possível aplicá-las para outras linguagens, uma vez que sua aplicação é independente do sistema e do microcontrolador utilizado. Atualmente a ferramenta *Doctor Who* aceita apenas arquivos do IDES (.xmd), no entanto, isso não é uma restrição, pois pretende-se futuramente atualizar a ferramenta para receber outros formatos de arquivos. Mostrou-se também como os arquivos gerados automaticamente pela ferramenta, bem como os arquivos que devem ser implementados pelo desenvolvedor, são alocados dentro da arquitetura de diagnóstico proposta no Capítulo 3. Por fim, constatou-se que a arquitetura de diagnóstico proposta, juntamente como a solução de geração automática de código, permitiram embarcar todo o software necessário para o diagnóstico de maneira rápida, localizada e sem interferência com o restante do software já existente do refrigerador, garantindo assim integridade dos algoritmos de controle originais. O capítulo seguinte é dedicado para a validação tanto dos modelos dos diagnosticadores como também da metodologia e da ferramenta de geração automática de código para implementação destes diagnosticadores.

6 VALIDAÇÃO DOS DIAGNOSTICADORES

No capítulo anterior mostrou-se que através da ferramenta *Doctor Who* foram gerados os arquivos necessários com o código em linguagem C, para cada diagnosticador projetado no Capítulo 4. Em seguida foram implementados os algoritmos necessários no módulo *Event.c*, para a geração dos eventos necessários para o diagnóstico. Uma vez implementados os diagnosticadores, é necessário validar a sua eficácia, ou seja, verificar se os modelos considerados para construí-los, bem como seu projeto estão corretos e também verificar se não foram introduzidos erros durante a fase de implementação do *software*, principalmente quando implementando o módulo *Events.c*. Neste capítulo serão mostrados as ferramentas usadas e desenvolvidas para a realização desta validação, bem como o processo de validação de cada um dos diagnosticadores anteriormente projetados.

6.1 Metodologia de Validação

Os arquivos de código C, gerados conforme mostrado no capítulo anterior para os diagnosticadores projetados no Capítulo 4, foram compilados juntamente com os demais arquivos que compõem o software principal do refrigerador utilizado no estudo de caso, e posteriormente embarcado no microcontrolador STM8S105K6 de 8 bits da família STM8, utilizado na placa de controle deste refrigerador.

Na validação do sistema de diagnóstico foi utilizado um refrigerador do mesmo tipo do mencionado no estudo de caso. Além disso, foram utilizadas duas ferramentas para auxiliar o processo de validação. Com o objetivo de monitorar e forçar algumas condições para se reproduzir algumas das falhas, foi utilizada uma ferramenta que permite o acesso tanto a variáveis e dados internos ao *software* bem como a manipulação destes dados em tempo real. Ainda, com a finalidade de prover uma interface

gráfica do estado de cada diagnosticador, facilitando assim o processo de validação, foi utilizada uma segunda ferramenta capaz de sincronizar os diagnosticadores executados dentro do *software* com seus respectivos modelos gerados pelo IDES.

Para a validação dos diagnosticadores relativos à camada de *software*, foram feitas algumas modificações pontuais no *software* principal do refrigerador a fim de forçar falhas nas rotinas para as quais os diagnosticadores haviam sido projetados. Já para a validação dos diagnosticadores relativos à camada de *hardware*, modificações em circuitos da placa de controle foram feitas a fim de reproduzir as condições das possíveis falhas que poderiam ocorrer. Finalmente, para a validação dos diagnosticadores da camada de cargas, foram simulados as condições reais de falha em cada uma das cargas.

A seguir será detalhado o funcionamento das ferramentas utilizadas e em seguida serão apresentados os detalhes referentes à validação de cada um dos diagnosticadores, bem como os resultados obtidos.

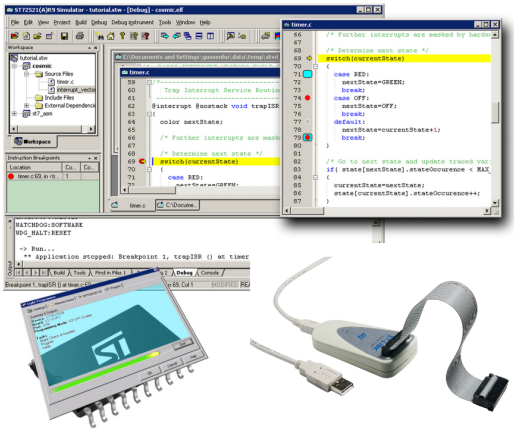
6.1.1 Ferramenta de Depuração

Para auxiliar a validação do software foi utilizada a ferramenta STVD (*ST Visual Develop*) (STVD, 2010). Esta ferramenta é uma IDE (*Integrated Development Environment*) fornecida pela STMicroelectronics para as famílias de microcontroladores ST7 e STM8, que permite o monitoramento e editação *online* de variáveis do *software* e também adicionar *software breakpoints* para fins de depuração, conforme ilustrado na Figura 6.1

6.1.2 Ferramenta Automata Player Monitor

Além do STVD também foi utilizada uma ferramenta chamada *Automata Player*. Esta ferramenta foi inicialmente desenvolvida em parceria entre Whirlpool e UDESC, voltada à aplicação de controle supervisório. Contudo, esta ferramenta possui um módulo de monitoramento

Figura 6.1: Ferramenta STVD



Fonte: Autor

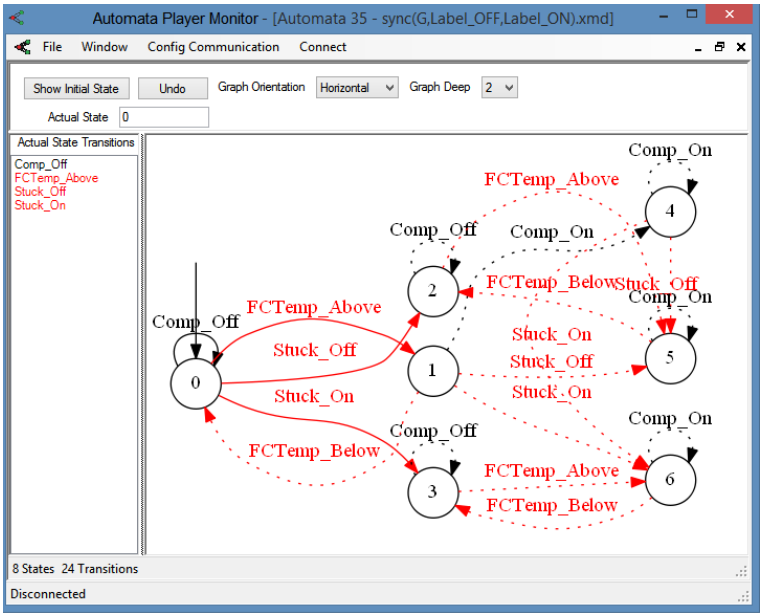
denominado *Automata Player Monitor*, que devido a sua generalidade do que diz respeito à interface de monitoração, foi possível utilizá-la também neste trabalho.

Esta ferramenta permite importar um autômato (arquivo no formato *.xmd* do IDES) e fornece uma interface visual que permite seguir *offline* e *online* a evolução dos estados deste autômato.

No modo *offline*, é possível disparar manualmente as transições de estados (eventos) e a ferramenta automaticamente atualiza o grafo de estados do autômato e o mostra na janela de interface. Já no modo *online* é necessário prover uma comunicação serial entre o PC (onde a ferramenta Automata Player Monitor está sendo executada) e o microcontrolador (onde os diagnosticadores estão implementadas) para assim acompanhar pela interface a evolução de estados que ocorre no sistema embarcado. O módulo *AutomataPlayer.c* já possui uma API que permite enviar por uma interface serial o estado atualizado de cada autômato.

Assim, quando no modo *online*, deve-se importar para esta ferramenta o mesmo arquivo *.xmd* usado para gerar os arquivos em linguagem C dos diagnosticadores. Logo, tanto a ferramenta como o microcontrolador terão os mesmos modelos dos autômatos diagnosticadores. Então, o algoritmo implementado na função *"AutomataPlayer__Handler"* atualizará os estados dos autômatos de acordo com a ocorrência de eventos, como discutido antes, e enviará via comunicação serial, usando a API definida, o estado atual de cada autômato diagnosticador. Finalmente, a ferramenta *Automata Player* irá receber esta mensagem e automaticamente atualizará o grafo de estados dos autômatos na janela de interface. A título de ilustração, na Figura 6.2 mostra-se um autômato exibido através desta ferramenta, cujo detalhes são descritos a seguir.

Figura 6.2: Ferramenta Automata Player Monitor



Fonte: Autor

O botão *Show Initial State* permite reiniciar o autômato para seu estado inicial. O botão *Undo* permite desfazer a última operação (evento), quando no modo *offline*. O campo *Graph Orientation* permite selecionar o modo de exibição do autômato entre horizontal e vertical. O campo *Graph Deep* permite selecionar a profundidade da alcançabilidade com qual o autômato deve ser exibido, isto é, quantos estados alcançáveis a partir do estado atual devem ser exibidos na janela de interface. O campo denominado *Actual State* fornece o estado atual do autômato. A barra localizada do lado esquerdo denominada *Actual State Transitions* mostra os eventos ativos para o estado atual, onde os eventos não controláveis são mostrados na cor vermelha e os eventos controláveis na cor preta. Quando operando no modo *offline*, é possível simular a ocorrência de qualquer uma destas transições clicando-se sobre a mesma. Na janela principal é mostrado o grafo do autômato, onde as transições referentes aos eventos não controláveis são mostradas em vermelho e as referentes aos eventos controláveis mostradas em preto. As linhas cheias representam as transições ativas para o estado atual, enquanto as tracejadas as transições dos estados futuros.

6.2 Validação dos Diagnosticadores: Detalhamento

Nesta seção será discutida a abordagem usada para simular e/ou forçar as diferentes falhas a fim de validar os diagnosticadores desenvolvidos. Dependendo da camada para qual o diagnosticador foi projetado, isto é camada de *Software*, *Hardware* ou Cargas, uma abordagem diferente foi utilizada, devido suas particularidades, conforme detalhado a seguir.

6.2.1 Validação dos Diagnosticadores da Camada de Software

Para validar os diagnosticadores desta camada, a abordagem utilizada foi simular (ou forçar) a falha através da introdução de *bugs* conhecidos em cada rotina de *software* para o qual foram desenvolvidos os diagnosticadores e assim verificar o resultado do diagnóstico.

A Figura 6.3 mostra um exemplo de um dos *bugs* propositais utilizados para simular a falha na rotina termostática. A introdução deste *bug* foi feita pela remoção da linha (a mesma foi comentada) responsável por mudar o estado da termostática para o estado *TC_COOLING*, isso fez com que a rotina termostática permanecesse no estado *TC_TEMP_SATISFIED* (este estado de *software* refere-se ao estado *TC_SATISF* do autômato), mesmo após a temperatura ficar acima o ponto de controle (*cutin*). Assim, a termostática manteve a solicitação para manter o compressor desligado, quando o mesmo deveria ser solicitado para ligar.

Para entender melhor este teste, considere que o diagnosticador da termostática (ver Figura 4.10) estava no estado {1N, 4Y} e que com o aumento da temperatura acima do ponto de controle tenha ocorrido o evento *Above_CP*, levando o autômato para o estado {2N, 4Y}. Nesta situação, era esperado que a termostática requisitasse o ligamento do compressor através do evento *Req_Comp_ON*, contudo devido ao *bug* inserido nesta rotina, a mesma permaneceu no *TC_TEMP_SATISFIED* requisitando o desligamento do compressor (evento *Req_Comp_OFF*), levando assim o autômato para o estado {4Y}, tornando-se assim certo da ocorrência da falha.

De forma semelhante, para simular uma falha na rotina de controle do compressor foi inserido um *bug* nesta rotina. Na Figura 6.4 mostra-se a mudança feita na linha *Compressor_Type[cmpr].State = COMPRESSOR_ON*, a qual foi modificada para *Compressor_Type[cmpr].State = COMPRESSOR_OFF* o que forçou o compressor ser desligado quando ele deveria na verdade ter permanecido ligado. O respectivo diagnosticador (ver Figura 4.14) estava inicialmente no estado {3Y, 6N}, que se refere ao estado WAIT_ON. Quando o tempo de proteção do compressor expirou, ou seja, a condição *Timers__HMSGetStatus(COMPRESSOR_PROTECTION_TIMER) == TIMERS_COMPLETED* foi satisfeita, o evento *CompOn_Prot_Timeout* foi

Figura 6.3: Inserção de Bug na Rotina Termostática para Validação do Diagnosticador

```
void Thermostatic__Handler(void)
{
    control_point =  NonVol.Thermo_Parameters[FREEZER].Control_Temperature;
    hysteresis = NonVol.Thermo_Parameters[FREEZER].Hysteresis;
    cutin = control_point + hysteresis;
    cutout = control_point - hysteresis;

    switch (Thermo_State[FREEZER])
    {    // state transitions
    case TC_TEMP_SATISFIED:
    case TC_COOLING:
        if (Temp_Control_Actual_Temperature[FREEZER] > cutin)
        {
            // Temp_Control_State[FREEZER] = TC_COOLING; //Change to Cooling
            if (Thermo_State[FREEZER] != TC_COOLING)
            {   //Set the Fan Delay Timer on transition
                Timers5s__Set(TIMER[EVAP_FAN_CONTROL_TIMER_COMPARTMENT_ONE],
                NonVol.Thermo_Parameters[FREEZER].Evap_Fan_Delay);
            }
        }
        else if (Temp_Control_Actual_Temperature[FREEZER] < cutout)
        {
            Thermo_State[FREEZER] = TC_TEMP_SATISFIED; //Change to Satisfied
            if (Thermo_State[FREEZER] != TC_TEMP_SATISFIED)
            {   // Set the Fan Extend Timer on transition
                Timers5s__Set(TIMER[EVAP_FAN_CONTROL_TIMER_COMPARTMENT_ONE],
                NonVol.Thermo_Parameters[FREEZER].Evap_Fan_Extend);
            }
        }
        break;
    case TC_OK_TO_DEFROST:
        if (!Defrost__GetReadyForDefrostFlag())
        {
```

Bug Proposital:
Removendo esta linha a Termostática nunca mudará para o estado *TC_COOLING*

Fonte: Autor

então gerado, mudando o diagnosticador para o estado {3Y, 7N}. Neste momento, o pedido para manter o compressor ligado ainda estava ativo, o evento *Req_Comp_On* deveria ter sido gerado e levado o autômato para o estado {3Y, 5N}, contudo devido a modificação feita nesta linha do código, o pedido foi para desligar o compressor, o que causou a ocorrência do evento *Set_Comp_Off*. Assim, o diagnosticador mudou para o estado {3Y} indicando por sua vez a ocorrência de falha.

Figura 6.4: Inserção de Bug na Rotina de Controle do Compressor para Validação do Diagnosticador

```
case COMPRESSOR_WAIT_ON:
   //Check Protection exit condition
   if(Timers__HMSGetStatus(COMPRESSOR_PROTECTION_TIMER) == TIMERS_COMPLETED)
   {
      if(Compressor_Type[cmpr].Request)  //If now the request ON still remains
      {
         //Compressor_Type[cmpr].State = COMPRESSOR_ON;     //Keep Compressor ON
         Compressor_Type[cmpr].State = COMPRESSOR_OFF;
      }
      else
      {
        //Compressor can be turned Off
        Compressor_Type[cmpr].State = COMPRESSOR_OFF;
        Timers__HMSSet(COMPRESSOR_PROTECTION_TIMER, 0, COMPRESSOR_MIN_OFF_TIME, 0);
      }
   }
   else
   {
      //Don't let compressor be OFF.
      if(Compressor_Type[cmpr].Request == OFF)
      {
         Compressor_Type[cmpr].Request = Compressor_Type[cmpr].Previous_Request;
      }
   }
   break;

case COMPRESSOR_OFF:
   if(Compressor_Type[cmpr].Request)
   {
      if(Timers__HMSGetStatus(COMPRESSOR_PROTECTION_TIMER) == TIMERS_RUNNING)
      {
```

Bug proposital:
Estado *COMPRESSOR_ON* mudado para *COMPRESSOR_OFF*

Fonte: Autor

A Figura 6.5 mostra um dos *bugs* inseridos para validar a rotina de degelo. Neste caso, o início de um novo degelo foi forçado enquanto um degelo ainda estava ocorrendo. A linha com *Defrost_Trigger = TRUE* foi inserida dentro do estado DROPPING da rotina de degelo. Assim, quando a rotina de degelo atingiu o estado DROPPING, a execução da linha *Defrost_Trigger = TRUE* causou o evento *Defrost_Trigger_True*,

fazendo o diagnosticador de degelo (mostrado na Figura 4.18) mudar do estado {3Y, 7N} para o estado {3Y}, que é um estado certo da ocorrência da falha.

Figura 6.5: Inserção de Bug na Rotina de Degelo para Validação do Diagnosticador

```
case DEFROST_STATE_EXTENDED:
    DEFROST_CONFIGURE_LOADS_EXTEND();
    DEFROST_SET_HEATER_ON();

    if(!Defrost_Time)
    {
        Defrost_State = DEFROST_STATE_DROPPING;
        Defrost_Time = defrost_parameters->Dropping_Time;
        One_Minute = 0;
    }
    break;

case DEFROST_STATE_DROPPING:
    DEFROST_CONFIGURE_LOADS_DROPPING();
    DEFROST_SET_HEATER_OFF();

    Defrost_Trigger = TRUE;

    if(!Defrost_Time)
    {
        Heat_On_Time = 0;
        Clock_Time = 0;
        Two_Minutes = 0;
        Compressor_Running_Time = 0;
        Compressor_Running_Time_Secs = 0;
        Compressor_Working_Factor = R_MAX;
        Defrost_State = DEFROST_STATE_MONITOR;
    }
    break;
```

Bug Proposital:
Adicionado evento de *Defrost Trigger* durante a execução do defrost

Fonte: Autor

6.2.2 Validação dos Diagnosticadores da Camada de Hardware

Os diagnosticadores dos relés detectam dois tipos de falhas: travado fechado e travado aberto. Para simular a situação de travado fechado, a entrada e a saída do relé foram curto-circuitadas. Já para simular a falha do relé travado aberto, a bobina do relé foi desconectada do circuito de acionamento, impedindo assim o seu acionamento e mantendo seus contatos sempre abertos.

No primeiro cenário, ou seja, relé travado fechado, alterando-se a temperatura para um valor acima do ponto de controle o compressor foi colocado no estado ligado, fazendo assim o diagnosticador ficar inicialmente no estado {(3N_FC N_FO),(2Y_FC N_FO),(4N_FC Y_FO)}. Em seguida, a temperatura foi alterada para um valor abaixo do ponto de controle, fazendo a termostática requisitar o desligamento do compressor. Contudo, o mesmo permaneceu ligado devido ao curto-circuito externo. Neste momento, o *feedback* do relé continuou indicando que o mesmo permanecia fechado, o que gerou o evento *FB_ON*. Por outro lado, o pedido da termostática para desligar o compressor gerou o evento *Set_Comp_OFF*. Estes dois eventos foram detectados pelo diagnosticador, devido ao mapeamento de sensor [*Set_Comp_OFF, FB_ON*], levando o diagnosticador para o estado {2Y_FC N_FO} o que indica que o diagnosticador é certo sobre a falha do tipo travado fechado (componente 2Y_FC) e, consequentemente, certo da não ocorrência da falha do tipo travado aberto (componente N_FO).

Para o segundo cenário (relé travado aberto) a temperatura foi inicialmente ajustada para um valor abaixo do ponto de controle de forma a manter o relé do compressor aberto. Neste momento, o diagnosticador estava no estado {(1N_FC N_FO),(2Y_FC N_FO),(4N_FC Y_FO)} como esperado. Em seguida, a temperatura foi alterada para um valor acima do ponto de controle, fazendo a termostática requisitar o ligamento do compressor, gerando então o evento *Set_Comp_On*. No entanto o relé permaneceu desligado, gerando por sua vez o evento *FB_Off*. Assim, es-

tes dois eventos foram detectados pelo mapeamento de sensor, levando o diagnosticador para o estado{4N_FC Y_FO}, ou seja, um estado onde o diagnosticador está certo sobre a ocorrência da falha do tipo travado aberto (componente Y_FO) e, consequentemente, certo da não ocorrência da falha do tipo travado fechado (componente N_FC).

O diagnosticador do relé da resistência de degelo foi validado utilizando a mesma abordagem, porém iniciando e terminando ciclos de degelo para forçar a abertura e fechamento deste relé.

6.2.3 Validação dos Diagnosticadores da Camada de Cargas

O modo de falha tanto para resistência de degelo como para o compressor é um circuito aberto, isto é, filamento de aquecimento rompido para o caso da resistência de degelo e bobina ou relé interno de partida danificado, no caso do compressor. Assim, ambas as falhas foram simuladas simplesmente deixando a respectiva carga desconectada da placa de controle.

O diagnosticador da resistência de degelo depende dos eventos C1, C2 e C3, que são uma combinação de eventos *HeaterRelay_FB_On*, *HeaterRelay_FB_Off*, *Def_End_By_Temp*, *Def_End_By_Timeout* e T_{ref}, como mapeados na Tabela 4.5. Na Figura 5.6 mostrou-se como os eventos C1, C2 e C3 são gerados pela combinação dos eventos acima citados. Assim, no início do degelo tanto a condição *Lsi__GetDigitalInput(HEATER_RELAY_FDBK) == RELAY_FEEDBACK_ON* quanto a condição *Defrost__GetDefrostState() == DEFROST_STATE_DEFROSTING* estavam satisfeitas, o que gerou o evento C2 (veja Figura 5.6). Em seguida, o degelo foi terminado por *timeout* e a temperatura do *Defrost Sensor* era menor que -7°C, conforme esperado, de acordo com o gráfico apresentado na Figura 4.39. Assim, as condições *Defrost__GetDefrostDuration() >= DEF_TIMEOUT* e *Sensor__GetValue(DEF_SENSOR) < (T_REF)* foram satisfeitas (Figura 5.6), o que gerou o evento C3, levando o diagnosticador para o estado certo de falha {3Y,4Y} (Figura 4.44).

O diagnosticador do compressor mostrado na Figura 4.35 depende do estado da porta, do sinal de *feedback* do relé do compressor e da variação positiva (delta positivo) da temperatura lida pelo sensor *Defrost Sensor*. A primeira avaliação feita neste diagnosticador foi mudando o estado da porta entre aberto e fechado, onde se observou na janela de interface da ferramenta *Automata Player*, que o diagnosticador estava mudando entre os estados {(3N,7Y),(5N,8Y)} e {(1N,4Y),(2N,6Y)} como esperado. Em seguida, mantendo a porta fechada, a temperatura foi ajustada para um valor acima do ponto de controle, forçando o compressor a ligar. Quando o relé do compressor fechou para ligar o compressor, seu *feedback* mudou para ligado também, gerando o evento *FB_ON*. No entanto, o compressor permaneceu desligado, pois o mesmo estava desconectado da placa de controle. Em seguida, a temperatura continuou se elevando gradativamente. Quando a temperatura aumentou mais de 5°C o evento de delta positivo foi gerado fazendo o diagnosticador mudar para o estado certo de falha {6Y,8Y,7Y,4Y}, diagnosticando assim a falha do compressor.

Posteriormente, foi repetido o mesmo teste anterior, entretanto desta vez a porta foi mantida aberta. Para essa condição a falha do compressor não poderia ser diagnosticada, uma vez que não seria possível distinguir se o aumento da temperatura era devido ao não ligamento do compressor ou à própria abertura da porta, conforme descrito anteriormente na Seção 4.5.1. Verificou-se então que o diagnosticador permaneceu no estado {(3N,7Y),(5N,8Y)}, devido à ocorrência do evento *Door_Opened* e mesmo tendo um delta positivo de temperatura, o diagnosticador não detectou falha. No entanto, após o fechamento da porta o diagnosticador mudou para o estado {(1N,4Y),(2N,6Y)} e assim que detectou o delta positivo de temperatura, ele finalmente muda para o estado {6Y,8Y,7Y,4Y}, indicando finalmente a ocorrência da falha.

6.3 Conclusões do Capítulo

Neste capítulo foi apresentada a metodologia e as ferramentas utilizadas para a validação da arquitetura multicamadas de diagnóstico. Os diferentes tipos de falhas foram simuladas através de alterações propositais no software, no hardware e nas cargas. Assim foi possível verificar a efetividade tanto dos modelos dos diagnosticadores como da implementação em software destes modelos, ou seja, validar tanto o código automaticamente gerado para o módulo *AutomataPlayer* como os algoritmos manualmente implementados no módulo *Event.c*, além de validar também a integração destes módulos com o software de controle do refrigerador já existente. A ferramenta *Automata Player Monitor* permitiu verificar de maneira visual e *online* a ocorrência dos eventos e a evolução dos estados autômatos diagnosticadores dentro do sistema embarcado. Já a ferramenta STVD auxiliou na depuração do software, permitindo o monitoramento e edição *online* de variáveis. Desta maneira, certificou-se que o software embarcado para o diagnóstico de falhas, conforme a arquitetura proposta, cumpriu os requisitos esperados do DAM (*Diagnoser Automata Model*, ver Figura 5.1), descritos na Seção 5.1 e também certificou-se efetividade do código automaticamente gerado pela ferramenta *Doctor Who*.

7 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi tratado o problema de diagnóstico de falhas preciso para sistemas embarcados. Para este problema foi proposto um sistema de diagnóstico multicamadas com foco nos principais elementos que compõem um sistema embarcado: *hardware*, *software*, cargas e sensores. O objetivo deste sistema é prover uma maior descriminação da origem da falha, tornando assim o diagnóstico mais preciso e possibilitando uma ação corretiva mais eficaz. Foram apresentadas também as condições necessárias para que se possa definir o sistema de diagnóstico como um sistema de diagnóstico multicamadas determinístico. Para que fosse possível embarcar este sistema de diagnóstico, foi proposta uma arquitetura que define as interfaces necessárias com o sistema embarcado (planta) onde se deseja realizar o diagnóstico. Esta arquitetura também considera as interfaces tanto para o gerenciamento como para recuperação das falhas. A arquitetura de diagnóstico proposta neste trabalho é independente tanto da metodologia usada para o projeto dos diagnosticadores quanto do tipo de microcontrolador onde a mesma será embarcada, possibilitando assim sua aplicação em diferentes tipos e modelagens de sistemas embarcados.

No estudo de caso de um refrigerador *Frost Free* apresentado neste trabalho, os diagnosticadores foram projetados no contexto de diagnose de falhas em sistemas a eventos discretos, conforme apresentado por Sampath et al. (1996). Para os autômatos diagnosticadores obtidos segundo esta metodologia, foi então desenvolvida uma classe de *software*, denominada DAM (*Diagnoser Automata Player*), composta pelas bibliotecas *Diagnoser Automata Player* e *Events* as quais permitiram embarcar e manipular adequadamente estes autômatos diagnosticadores. Na biblioteca *Diagnoser Automata Player* os autômatos diagnosticadores são representados por meio de tabelas padronizadas e configurados através de um arquivo de parametrização desta biblioteca (*Automata-*

Player_prv.h). Contudo a tradução do autômato para linguagem de código através da configuração destas tabelas não está livre de erros, uma vez que este é um processo manual. Para se resolver este problema, neste trabalho também foi desenvolvida uma ferramenta denominada *Doctor Who*, a qual permite a geração automática de código para essas tabelas, através da importação do arquivo .xmd gerado pelo programa IDES, usado para modelar o autômato diagnosticador. Esta ferramenta também gera os demais arquivos de código da biblioteca *Diagnoser Automata Player*, isto é, os aquivos *AutomataPlayer.c* e *AutomataPlayer.h* além da lista de eventos necessários para o diagnóstico, no arquivo *Events.h*.

Assim, neste trabalho foi apresentada uma solução completa para o diagnóstico em sistemas embarcados, deste a arquitetura de diagnóstico, a qual provê as interfaces necessárias para a integração com o sistema embarcado; o sistema de diagnóstico multicamadas que permite um diagnóstico mais preciso que o usual; uma metodologia de implementação em *software* dos diagnosticadores, quando os mesmos são projetados do ponto de vista de SEDs e por fim uma ferramenta para automaticamente gerar este *software*.

Como possíveis continuações deste trabalho, podem-se citar: *(i)* o desenvolvimento do sistema de recuperação (*Self Recovery*) ou tolerante à falha (*Fault-Tolerant*) em SEDs, o qual pode ser incorporado na arquitetura proposta neste trabalho; *(ii)* a utilização de diagnosticador robusto para a identificação dos sensores que falharam durante o processo de diagnose de falhas e *(iii)* utilizar outra metodologia para o projeto do diagnosticadores, tal como redes de Petri, a fim de se comparar a complexidade e tamanho do software necessário para embarcar os diagnosticadores.

REFERÊNCIAS BIBLIOGRÁFICAS

ATHANASOPOULOU, E., LINGXI, L. e HADJICOSTIS, C. (2010). **Maximum likelihood failure diagnosis in finite state machines under unreliable observations**, IEEE Transactions on Automatic Control, 2010.

BALEMI, S. **Control of Discrete Event Systems: Theory and Applications**, Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, 1992.

BARR, M. **Programming Embedded Systems, With C and GNU Development Tools**. 2nd Edition, O'Reilly, 2006, p.21.

BASILIO, J. C. e LAFORTUNE, S. **Robust codiagnosability of discrete event systems**, Proceedings of the American Control Conference, St. Louis, Missouri, 2009, p. 2202–2209.

BASILIO, J. C. ,CARVALHO, L. K. E MOREIRA, M. V. **Diagnose de falhas em sistemas a eventos discretos modelados por automatos finitos**, Revista Controle & Automação (Impresso), v. 21, p. 510-533, 2010.

BASILIO, J. C., LIMA, S. T. S., LAFORTUNE, S., e MOREIRA, M. V. **Computation of minimal event bases that ensure diagnosability**. Discrete Event Dynamic Systems, v. 22, p. 249-292, 2012.

CARVALHO, L. K., BASILIO J.C., MOREIRA, M.V. **Robust diagnosability of discrete event systems subject to intermittent sensor failures**. In: Proc. of 10th international Workshop on Discrete Event Systems, Berlin, Germany, 2010, p. 94–99

CARVALHO, L. K. **Diagnose Robusta de Sistemas a Eventos Discretos**. Tese (Doutorado em UFRJ COPPE-PEE Programa de Engenharia Elétrica) - Universidade Federal do Rio de Janeiro, 2011.

CARVALHO, L. K. BASILIO, J. C. , e MOREIRA, M. V. **Robust diagnosis of discrete event systems against permanent loss of observations**. Automatica. Vol. 49, Vol. 1, January 2013, p. 223-231.

CASSANDRAS, C. G., LAFORTUNE, S. **Introduction to Discrete Event Systems**, 2nd ed. Boston, Kluwer Academic Publishers, 2008.

CONTANT, O., LAFORTUNE, S., TENEKETZIS D. **Diagnosability of Discrete Event Systems with Modular Structure**. Discrete Event Dynamic Systems, 2006, 16: 9–37

CURY, J. E. **Teoria de Controle Supervisório de Sistemas a Eventos Discretos**. V Simpósio de Automação Inteligente, 2001.

DEBOUK, R., LAFORTUNE, S. e TENEKETZIS, D. **Coordinated decentralized protocols for failure diagnosis of discrete event systems**, Discrete Event Dynamic Systems: Theory and Applications 10: 33–86, January, 2000.

_____. Embarcados 2014. **Introdução: Embarcados Básico**. Disponível em: <http://www.embarcadosbasico.wordpress.com>. Acesso em: 15 de Agosto de 2014.

HALL, ELDON C. **Journey to the Moon: The History of the Apollo Guidance Computer**. Reston, Virginia, USA: American Institute of Aeronautics and Astronautics, 1996. p. 196.

HALGAMUGE, S. **Advanced Methods for Fusion of Fuzzy System and Neural Networks in Intelligent Data Processing**. Fortschr. Ber. VDI Reihe 10, Nr. 401, VDI-Verlag - Diisseldorf, 1996.

HOPCROFT, J. E., MOTWANI, R., ULLMAN, J. D. **Introduction to automata theory, languages, and computation**. 3 ed. Boston, Prentice Hall, 2006.

IDES (Integrated Discrete-Event Systems) Release 3 beta 1, September 2010 Disponível em: <https://qshare.queensu.ca/Users01/rudie/www/software.html>. Acesso em: 20 de Fevereiro, 2014.

ISERMANN, R. **Supervision fault-detection and fault-diagnosis methods-an introduction**, Control Eng. Practice, vol. 5, no. 5, pp 639-652, 1997.

JESUS, T., C. **Diagnosticabilidade e Diagnose On-Line de Falhas de Sistemas a Eventos Discretos Modelados por Autômatos Finitos**. Dissertação (Mestrado em UFRJ COPPE-PEE Programa de Engenharia Elétrica) - Universidade Federal do Rio de Janeiro, 2011.

JIANG, S., HUANG, Z., CHANDRA, V. e KUMAR, R. **A polynomial algorithm for testing diagnosability of discrete-event systems**, IEEE Transactions on Automatic Control, 2001, 46(8): 1318–1321.

JIROVEANU, G., BOEL, R.K., e DE SCHUTTER, B. **Fault diagnosis for time Petri nets**, In Proc. WODES 06, Ann Arbor - USA, Julho 2006.

KERNIGHAN, B.; RITCHIE, D. **The C Programming Language**. 2nd. ed. Englewood Cliffs,New Jersey,USA: Prentice Hall Inc., 1988.

LAFORTUNE, S., TENEKETZIS, D., SAMPATH, M., SENGUPTA, R., e SINNAMOHIDEEN, K. **Failure diagnosis of dynamic systems: An approach based on discrete event systems**. Proceedings of the American Control Conference, Arlington, USA, p. 2058-2071, 2001.

LIEM C. e PAULIN P. **Compilation techniques and tools for embedded processor architectures**, Chapter 5. In Hardware/Software Co-Design: Principles and Practice, eds. J. Staunstrup and W.Wolf, Boston: Kluwer Academic Publishers, 1998.

LIMA, S. T. **Diagnose Robusta de Sistemas a Eventos Discretos Sujeitos à Perda Permanente de Sensores**. Dissertação (Mestrado em UFRJ COPPE-PEE Programa de Engenharia Elétrica) - Universidade Federal do Rio de Janeiro, 2010.

LIMA, S,T., BASILIO, J.C., LAFORTUNE, S., MOREIRA, M.V. **Robust diagnosis of discrete-event systems subject to permanent sensor failures**. In: Proc. of 10th International Workshop on Discrete Event Systems, Berlin, Germany, 2010, p. 100–107, 2010a.

LIMA, S,T., BASILIO, J.C., LAFORTUNE, S., MOREIRA, M.V. **Bases Mínimas para a Diagnose de falhas de Sistemas a Eventos discretos. Parte 1: Eventos Essenciais para a diagnose e Trajetórias Primas**. XVIII Congresso Brasileiro de Automática, Setembro 2010, Bonito-MS, 2010b.

LIN, F. **Diagnosability of discrete event systems and its applications**, Journal of Discrete Event Dynamic Systems, v. 4, p. 197–212, 1994.

LIN, W. C.,Garcia, H. E., Yoo, T. S. **A diagnoser algorithm for anomaly detection in DEDS under partial and unreliable observations: characterization and inclusion in sensor configuration optimization**. Discrete Event Dynamic Systems (2013) 23:61–91.

Lunze, J., Schröder, J. (2004). **Sensor and actuator fault diagnosis of systems with discrete inputs and outputs**, IEEE Trans. Systems, Man and Cybernetics, Part B, Vol.34, n. 3, 2004, p. 1096-1107.

MAAS, D. N., LEAL, A. B., PINOTTI, A. J. **Síntese e Implementação de Controle Supervisório Monolítico para um Ice Maker**. XIX Congresso Brasileiro de Automática (CBA), Setembro 2012, Campina Grande-PB.

MOREIRA, M. V., JESUS, T. CARVALHO, K. L., BASILIO J. C. **Um novo Algorítimo para Verificação da Diagnosticabilidade Descentralizada de Sistemas a Eventos Discretos**. XVIII Congresso Brasileiro de Automática (CBA), Setembro 2010, Bonito-MS.

QIU W.; KUMAR, R. **Decentralized Failure Diagnosis of Discrete Event Systems**, IEEE Transactions on Systems, Man and Cybernetics- Part A: Systems and Humans, 2005.

RAMADGE, P. J.; WONHAM, W. M. **The Control of Discrete Event Systems**, Proceedings of IEEE,Vol. 77, n. 1, 1989, p. 81-98.

RIVIERA, M. H. M. **Diagnóstico de Falhas em Sistemas a Eventos Discretos: Uma Proposta de Aplicação em Processos de Separação Óleo-Gás**, Dissertação de Mestrado, COPPE/UFRJ, 2007.

ROHLOFF, K. R. **Sensor failure tolerant supervisory control**, Proc. and 2005 European Control Conference Decision and Control CDC-ECC '05. 44th IEEE Conference on, Seville, Spain, 2005, p. 3493–3498.

SAMPATH M., SENGUPTA, R., LAFORTUNE, S. **Diagnosability of discrete-event systems**, IEEE Trans. on Automatic Control, v. 40, 1995, p. 1555–1575.

SAMPATH M., SENGUPTA, R., LAFORTUNE, S. **Failure Diagnosis Using Discret-Event Models**, IEEE Trans. on Automatic Control Systems Technology, vol 4, 1996, p. 105–124.

STVD (ST Visual Develop) IDE for developing ST7 and STM8 MCUs applications, Setembro 2010 Disponível em: <http://www.st.com/web/catalog/tools/FM147/CL1794/SC1808/SS1767/PF210567> . Acesso em: 15 de Agosto de 2014.

TEIXEIRA, E. **Diagnóstico Inteligente de Falhas em um Processo de Separação Óleo-Gás em Plataformas Offshore**, Dissertação de Mestrado, COPPE/UFRJ, 1993.

WANG, Y., YOO, T. S. e LAFORTUNE, S. **Diagnosis of Discrete Event Systems using Decentralized Architectures**, Discrete, Event Dynamic Systems: Theory and Applications , Vol. 17, No. 2, June 2007, p. 233-263.

WOLF, W., **Computers as Components Principles of Embedded Computing System Design**, Second Edition, Morgan Kaufmann Publishers 2008, Chapter 1.

ZAD, S. H., KWONG, R. H. e WONHAM, W. M. **Fault diagnosis in discrete-event systems: framework and model reduction**, IEEE Transactions on Automatic Control, 2003, p. 1199–1212.

ZOETEWEIJ, P., PIETERSMA, J., ABREU, R., FELDMAN, A. e GEMUND, A. J. C. **Automated Fault Diagnosis in Embedded Systems**, The Second International Conference on Secure System Integration and Reliability Improvement, 2008