UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Günther Sgandella Klüsener

Uso de teste baseado em aprendizagem para a validação de programas de clp na indústria de petróleo e gás natural

Florianópolis
2020

Günther Sgandella Klüsener

Uso de teste baseado em aprendizagem para a validação de programas de clp na indústria de petróleo e gás natural

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina para a obtenção do título de Mestre em Engenharia de Controle e Automação
Orientador: Prof. Max Hering de Queiroz, Dr.
Coorientadores: Prof. Fabio Luis Baldissera, Dr. e Rodrigo Tacla Saad, Dr.

Florianópolis
2020

Ficha de identificação da obra

Klüsener, Günther Sgandella
Uso de teste baseado em aprendizagem para a validação de programas de clp na indústria de petróleo e gás natural / Günther Sgandella Klüsener ; orientador, Max Hering de Queiroz, coorientador, Fabio Luis Baldissera, coorientador, Rodrigo Tacla Saad, 2020.
80 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2020.

Inclui referências.

1. Engenharia de Automação e Sistemas. 2. Validação de software. 3. Métodos formais. 4. Model checking. 5. Teste de conformidade. I. Queiroz, Max Hering de . II. Baldissera, Fabio Luis. III. Saad, Rodrigo Tacla IV. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Engenharia de Automação e Sistemas. V. Título.

Günther Sgandella Klüsener

Uso de teste baseado em aprendizagem para a validação de programas de clp na indústria de petróleo e gás natural

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Eng. Marcelo Lopes de Lima, Dr.
CENPES - Petrobrás

Prof. Eric Aislan Antonelo, Dr.
Universidade Federal de Santa Catarina

Prof. Felipe Gomes de Oliveira Cabral, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em engenharia de controle e automação.

Documento assinado digitalmente
Werner Kraus Junior
Data: 09/01/2021 18:13:27-0300
CPF: 531.085.239-53

Coordenação do Programa de Pós-Graduação

Documento assinado digitalmente
Max Hering de Queiroz
Data: 23/12/2020 17:06:24-0300
CPF: 912.550.079-15

Prof. Max Hering de Queiroz, Dr.
Orientador

Florianópolis, 2020.

Este trabalho é dedicado à minha família, aos meus professores e amigos.

AGRADECIMENTOS

Agradeço, primeiramente, à minha noiva Marina Maia e aos meus pais Adalberto Alfredo Klüsener e Juçara Joana Sgandella Klüsener pela imensurável contribuição nos bastidores deste trabalho, motivacional e financeiramente.

Ao meu orientador Prof. Dr. Max Hering de Queiroz e aos coorientadores Prof. Dr. Fabio Luis Baldissera e Dr. Rodrigo Tacla Saad pela primorosa condução do presente trabalho e pelos ensinamentos.

A todos familiares e amigos que, de alguma forma, lembraram-se de mim e incentivaram-me durante a execução deste trabalho.

Agradecimentos à CAPES pelo suporte financeiro. Aos professores, colegas e funcionários do Departamento de Automação e Sistemas da UFSC pelo ambiente produtivo instaurado no entorno desta dissertação.

“Nada é bom ou mau, nosso pensamento é que o faz. ”

(William Shakespeare)

RESUMO

A exploração de Petróleo e Gás Natural é uma atividade complexa e de alto risco. Por este motivo a segurança nesta indústria é organizada em camadas de proteção independentes e regidas por normas. Dentre estas camadas estão os Sistemas Instrumentados de Segurança (SIS), a última alternativa para a extinção de acidentes. Esse sistema automático, composto por um Controlador Lógico Programável (CLP), sensores e atuadores deve ser testado a fim de identificar falhas em sua lógica de atuação. Esta dissertação apresenta um método automático para teste de conformidade baseado em especificação utilizando princípios de aprendizagem computacional. Os casos de teste são gerados e testados no CLP iterativamente. O comportamento do CLP frente aos testes é modelado iterativamente por um algoritmo de aprendizado. A partir das especificações de segurança descritas na forma de Matriz de Causa e Efeito, são extraídas fórmulas de lógica proposicional. Estas fórmulas são verificadas no modelo por um algoritmo de *model checking*. Eventualmente, contraexemplos podem ser encontrados, que representam inconformidades apenas do modelo ou também da lógica implementada no CLP. Os contraexemplos são executados como testes e um processo decisório, chamado de oráculo, compara a saída do CLP com o modelo, fornecendo o veredicto a respeito da inconformidade. O algoritmo encerra sua execução ao encontrar uma falha no sistema ou por critério de parada. Essa técnica é aplicada a um modelo simplificado. A técnica proposta identificou a inconformidade contida no modelo simplificado, inconformidade esta não detectada através da execução do conjunto de testes gerados por um método tradicional.

Palavras-chave: Teste baseado em Aprendizagem. *Model Checking*. Teste de Conformidade. Validação de CLP. Métodos Formais. Sistemas Instrumentados de Segurança. Indústria de Petróleo e Gás. Angluin L*. IDS. Moore MI.

ABSTRACT

The exploration of Oil and Natural Gas is a complex and high risk activity. For this reason, security in this industry is organized into independent layers of protection and are governed by standards. Among these layers are the Safety Instrumented Systems (SIS), the last alternative for the extinction of accidents. This automatic system, composed of a Programmable Logic Controller (PLC), sensors and actuators must be tested in order to identify flaws in the operating logic. This dissertation presents an automatic method for conformity testing based on specification using computational learning principles. Test cases are generated and tested on the PLC iteratively. The behavior of the PLC in relation to the tests is modeled iteratively by a learning algorithm. From the security specifications described in the form of Cause and Effect Matrix, formulas of propositional logic are extracted. These formulas are verified in the model by a model checking algorithm. Eventually, counterexamples can be found, which represent nonconformities only in the model or also in the logic implemented in the PLC. Counterexamples are performed as tests and a decision-making process, called an oracle, compares the output of the PLC with the model, providing the verdict regarding the non-conformity. The algorithm ends its execution when it finds a fault in the system or due to stopping criteria. This technique is applied to a simplified model. The proposed technique identified the non-conformity contained in the simplified model, which was not detected through the execution of the set of tests generated by a traditional method.

Keywords: *Learning-based Testing. Model Checking. Conformance Test. PLC Validation. Formal Methods. Safety Instrumented Systems. Oil and Gas Industry. Angluin L*. IDS. Moore MI.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CLP Controlador Lógico-Programável

SIS Sistema Instrumentado de Segurança

IPG Indústria de Petróleo e Gás Natural

SUT *System Under Test*

LBT *Learning-based Testing*

GCT Gerador de Casos de Teste

ADEF Autômato Determinístico de Estados Finitos

IDS *Incremental Distinguishing Sequences*

TES Traço Entrada-Saída

PTA *Prefix Tree Acceptor*

MCE Matriz de Causa e Efeito

FD *Fail on Demand*

ST *Spurious Trip*

LTL Lógica Temporal Linear

A1 Atuador

S1 Sensor 1

S2 Sensor 2

M1 Memória

F1 *Flip Sensor 1*

F2 *Flip Sensor 2*

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

A Indústria de Petróleo e Gás Natural (IPG) é constituída por uma série de processos de alto risco, os quais podem ser resumidos em: perfuração de poços, extração de hidrocarbonetos, separação de seus componentes e logística. No Brasil, diferentemente da exploração global de petróleo, a qual desde 2005 concentram aproximadamente 70% da produção em poços continentais (EIA, 2016), a exploração de campos marítimos correspondeu a 95,7% do volume total de petróleo produzido no país em março de 2019 (ANP, 2019).

Sendo assim, devido a tamanha complexidade da IPG e ao potencial impacto de acidentes associados às atividades desta indústria, os Sistemas Instrumentados de Segurança (SIS) são de extrema importância, uma vez que são a última camada de prevenção de acidentes. Estes, por sua vez, são compostos por Controladores Lógico-Programáveis (CLP) e instrumentos instalados com o propósito de conduzir o processo ou específico equipamento do processo para um estado seguro. Tais sistemas são projetados para atuarem apenas em condições perigosas e responsabilizam-se pela geração de saídas corretas para prevenção ou mitigação do acidente (GRUHN; CHEDDIE, 2006).

É importante ressaltar que a IPG utiliza a Matriz de Causa e Efeito (MCE) como documento padrão para definir os requisitos necessários dos SIS e esta, por sua vez, é regida pela norma IEC-62881 (2018), a qual formaliza os requisitos operacionais da construção da MCE. Além disso, tem-se a norma IEC-61511 (2018) que padroniza os requisitos funcionais de segurança dos SIS para a indústria de processos e apresenta boas práticas de projeto para *hardware* e *software* destes sistemas. Na norma, é previsto o uso de métodos formais e semiformais para validação de SIS, procedimento este utilizado para confirmar que o sistema instalado e comissionado cumpre os requisitos de segurança.

Com relação a isso, são utilizados métodos para validar seu correto funcionamento, dentre eles o *model checking*, Teste baseado em Aprendizagem e o teste de conformidade, sendo este último utilizado na IPG. O procedimento de testes, diferentemente da verificação formal, não é sempre realizado de maneira exaustiva, pois a explosão combinatória impede que todos os possíveis testes sejam realizados sob um determinado sistema. O uso de métodos formais, como *model checking*, também elemento de estudo do presente trabalho, fornece uma resposta absoluta quanto à corretude do modelo do sistema implementado. No entanto, necessita de uma modelagem congruente com o comportamento do sistema, o que para o contexto em questão pode não ser realizável.

Dessa limitação, surgiram novas pesquisas voltadas para técnicas de construção algorítmica de modelos de engenharia a partir de sistemas. A ideia de combinar as áreas de aprendizado de máquina e de teste foi enunciada em 1983 por Weyuker (BENNACEUR; MEINKE, 2018). Contudo, apenas com as recentes evoluções de algoritmos de aprendizado de autômatos e de *model checking*, que esta abordagem prática se tornou escalável para problemas industriais. Uma arquitetura apresentada em (MEINKE; NIU; SINDHU, 2012) propõe a utilização de métodos de aprendizagem de máquina para a realização de testes comportamentais mais eficientes e de maneira automática.

Esta abordagem, chamada Teste Baseado em Aprendizagem, é objeto de estudo deste trabalho. Adaptações no método foram necessárias para viabilizar sua aplicabilidade em Sistemas Instrumentados de Segurança. A fim de permitir a utilização da linguagem de domínio da aplicação para os requisitos de segurança, propomos incorporar a metodologia de tradução destes requisitos para fórmulas lógicas apresentada por Dos Reis (2018). A partir dos algoritmos de inferência de modelos estudados, propomos um novo procedimento de completar autômatos determinísticos que explicita os estados ainda não testados no sistema.

Sendo assim, dada a importância dos Sistemas Instrumentados de Segurança (SIS) e seus métodos de validação, este trabalho torna, como objeto de estudo, a técnica de Teste Baseado em Aprendizagem, juntamente ao âmbito de pesquisa da Universidade Federal de Santa Catarina com a Petrobras.

1.1 OBJETIVOS

O objetivo principal desta dissertação consiste em aplicar o método de Teste Baseado em Aprendizagem para a validação de softwares de CLP de Sistemas Instrumentados de Segurança empregados na Indústria de Petróleo e Gás Natural.

Dentre os objetivos específicos deste trabalho estão a especificação dos requisitos de software para que a técnica de Teste baseado em Aprendizagem seja adotada como parte da cadeia de verificação de programas de CLP na Petrobras; analisar e comparar a técnica de Teste Baseado em Aprendizagem com outras técnicas de validação; aplicar o método a um exemplo prático de SIS desenvolvido no projeto.

1.2 ESTRUTURA DO DOCUMENTO

Esta dissertação apresenta no próximo capítulo os sistemas de segurança na indústria de petróleo de gás e a metodologia adotada pela Petrobras no desenvolvimento de *software* de Sistemas Instrumentados de Segurança. Duas alternativas, no estado da arte, para a validação destes sistemas são apresentadas ainda no Capítulo 2.

No Capítulo 3, detalhamos a técnica Teste Baseado em Aprendizagem, apresentando inicialmente a arquitetura e seu procedimento algorítmico. Na sequência focamos nosso estudo no componente central da técnica, o algoritmo de aprendizado. Apresentamos didaticamente três algoritmos de aprendizado de autômatos aplicando-os a um caso simplificado. Concluímos o Capítulo 3 analisando o desempenho destes algoritmos, e elencando a abordagem a ser utilizada em nossa proposta de Teste Baseado em Aprendizagem para verificação da conformidade de Sistemas Instrumentados de Segurança.

No Capítulo 4, apresentamos nossa proposta detalhando e justificando as alterações feitas no método proposto por Meinke. Desenvolvemos um modelo simplificado, com erro de implementação, que preserva as características de memória presentes em Sistemas Instrumentados de Segurança. Comparamos o método proposto a um algoritmo tradicional de geração de casos de teste aplicando-os ao modelo desenvolvido.

No Capítulo 5, concluímos o trabalho compilando os estudos realizados, nossa proposta de Teste Baseado em Aprendizagem e resultados obtidos para a proposição de trabalhos futuros.

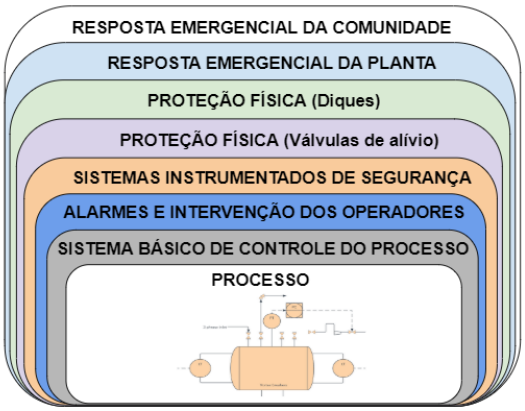
2 DESCRIÇÃO DO PROBLEMA

O problema da identificação de falhas na implementação de *softwares* de Controladores Lógico-Programáveis (CLP) integrantes dos Sistemas Instrumentados de Segurança (SIS) em plataformas exploratórias de petróleo é uma questão crítica frente à proporção que acidentes podem tomar em termos ambientais e socioeconômicos. No presente capítulo apresentamos a organização dos sistemas de segurança na Indústria de Petróleo e Gás Natural, a metodologia de desenvolvimento de SIS adotada atualmente pela Petrobras e duas alternativas para validação de tais sistemas no estado da arte.

2.1 SISTEMAS DE SEGURANÇA NA IPG

A Indústria de Petróleo e Gás Natural adota uma arquitetura de segurança organizada em camadas de funções específicas e independentes entre si. À medida em que uma camada de segurança não é capaz de extinguir o perigo, a próxima camada entra em atuação e assim por diante, aumentando, desta forma, a confiabilidade do sistema. A seguir discorremos brevemente sobre cada uma das camadas, conforme ilustrado na Figura 1.

Figura 1 – Camadas de proteção na indústria de processos.



Fonte: Adaptado de (GHUN; CHEDDIE, 2006).

A prevenção de acidentes inicia-se no projeto de uma planta de processo intrinsecamente segura. O Sistema Básico de Controle do Processo (*Basic Process Control System* – BPCS) é a próxima camada de segurança, responsável por manter todas as variáveis do processo em valores seguros. Contudo, uma falha no BPCS pode iniciar um evento perigoso. Caso isso ocorra, alarmes devem ser utilizados para alertar os operadores de que alguma

intervenção da parte deles é requerida. A não solução do problema nesta camada deve ser detectada pelo SIS, que atuará na prevenção ou mitigação desta condição de perigo. O insucesso do SIS resulta no acidente.

Ocorrido o acidente, as camadas de mitigação devem reduzir as consequências. Primeiramente, através das camadas de Proteção Física, geralmente compostas por válvulas de alívio, sistemas de Fogo e Gás, depuradores, *flares* e sistemas de contenção que são uma proteção da planta. Quando o acidente não é mitigado pelas camadas de proteção física, procedimentos de evacuação devem ser adotados para retirar as pessoas da planta e/ou de uma área sujeita a esse acidente, representado pelas duas camadas mais externas (ver Figura 1).

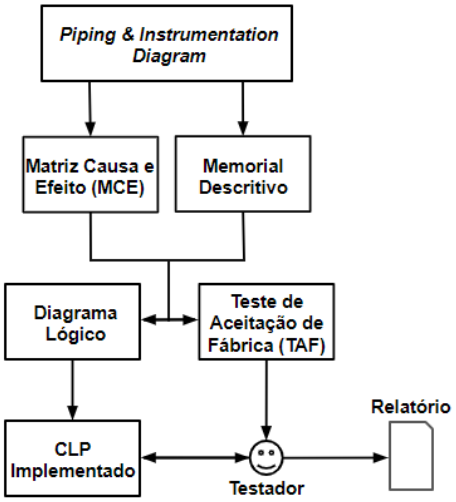
2.2 METODOLOGIA DE DESENVOLVIMENTO DE SIS

O desenvolvimento do *software* da camada SIS na Petrobras é regido por normas internas (N-1883, 2002) (N-2595, 2012), que definem a geração de diversos documentos padronizados (Figura 2). Se inicia com a análise de riscos do processo, a partir da qual é criado o Diagrama de Tubulação e Instrumentação (*Piping and Instrumentation Diagram* - P&ID), onde são definidas a instrumentação e as malhas de controle do sistema. A lógica de atuação em caso de falhas também é derivada da análise de riscos e, posteriormente, documentada na forma de Matriz de Causa e Efeito (MCE), uma tabela que especifica a relação entre os sensores (causas) e os atuadores (efeitos). As sequências de inicialização e desligamento do Sistema de Controle Básico do Processo (BPCS) são descritas em um documento denominado Memorial Descritivo.

Na sequência, os requisitos do programa e um plano de teste são documentados por um Diagrama Lógico e um Teste de Aceitação de Fábrica (TAF). O Diagrama Lógico representa as lógicas de operação do sistema conforme a norma ISA 5.2 e o TAF lista as funções de segurança a serem testadas.

A implementação do *software* é realizada a partir do diagrama lógico por uma empresa terceirizada. Contudo, a verificação da conformidade com o requisito é efetuada pela Petrobras através da execução do TAF. As inconsistências identificadas são, então, listadas em um relatório para que as futuras alterações sejam realizadas. Quando não forem identificadas inconsistências, o dispositivo é então liberado para entrega e instalação na planta (VEIGA, 2018).

Figura 2 - Processo de desenvolvimento de SIS na Petrobras.



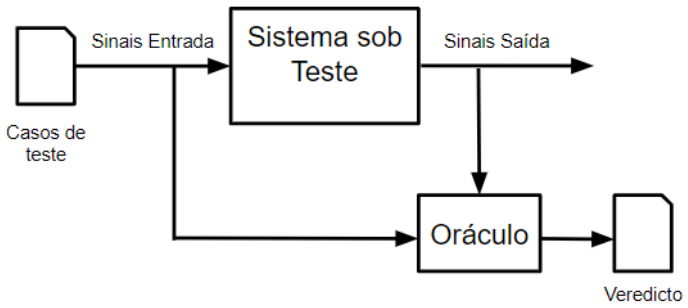
Fonte: Adaptado de (VEIGA, 2018).

Os métodos de verificação e de validação de *software*, quando aplicados apropriadamente ao ciclo de vida do projeto, podem resultar em programas de maior qualidade e confiabilidade (WALLACE; FUJII, 1989). A verificação envolve a avaliação do *software* durante cada fase do ciclo de vida para garantir que este cumpra os requisitos definidos na fase anterior. A validação envolve teste de *software* ou de suas especificações ao final da etapa de desenvolvimento para garantir que o mesmo cumpre seus requisitos. Apresentamos a seguir algumas características do método de teste de conformidade.

2.3 TESTE DE CONFORMIDADE

O teste de conformidade é um método de validação de sistemas, gerando-se a partir da especificação um conjunto de casos a serem testados diretamente no CLP com o *software* a ser validado (*System Under Testing* - SUT). Um dispositivo, chamado Oráculo, é responsável por avaliar as saídas do SUT para os casos testados e fornecer um veredicto quanto à corretude do comportamento do sistema, conforme ilustrado na Figura 3 (GERGELY; COROIU; POPENTIU-VLADICESCU, 2011).

Figura 3 - Abstração do procedimento de teste de conformidade de sistemas.



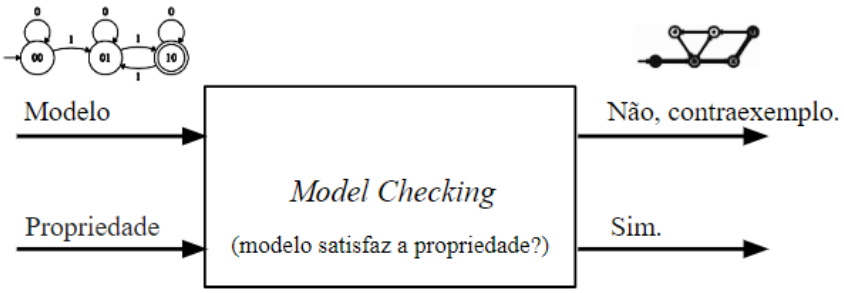
Fonte: Elaboração do autor (2020).

Uma vantagem do teste de conformidade em relação à verificação formal é o fato de que ele é aplicado sobre o sistema real e, assim, permite inferir propriedades do sistema real e não de um modelo matemático de tal sistema, conforme detalhamos a seguir.

2.4 MODEL CHECKING

A verificação por *model checking* é um procedimento que permite determinar se um modelo matemático satisfaz uma propriedade expressa em fórmula de lógica temporal (CLARKE et al, 2009). Esta técnica visita exaustivamente todo o espaço de estados do modelo, aferindo a satisfação dos requisitos. O *model checking* possui algumas vantagens quando comparado a outros métodos. Primeiramente, executa uma verificação exaustiva que fornece ao usuário a certeza da satisfação ou não da propriedade investigada. Outra importante vantagem é a apresentação de um contraexemplo caso a propriedade não seja satisfeita, conforme ilustrado na Figura 4, que auxilia o projetista a encontrar o erro.

Figura 4 – Abstração do procedimento de *model checking*.



Fonte: Elaboração do autor (2020).

O fato de ser um procedimento algorítmico realizado sobre um modelo viabiliza também sua utilização em etapas anteriores à implementação, auxiliando na identificação precoce de erros. No entanto, a dificuldade na obtenção de um modelo matemático e o problema da explosão do espaço de estados podem limitar a utilização do *model checking*. Importante ressaltar também que um modelo é uma abstração do sistema, por este motivo a técnica deve ser utilizada com ressalvas.

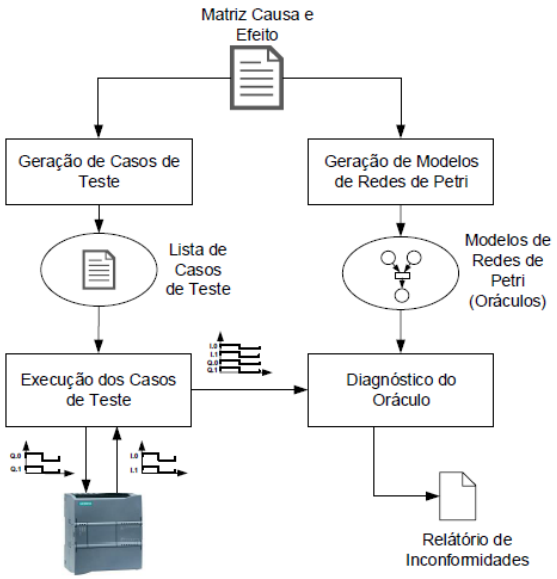
Apresentamos, na sequência, as abordagens de teste de conformidade e de verificação formal por *model checking* aplicadas a SIS de plataformas de petróleo propostas, respectivamente, por Veiga (2018) e Dos Reis (2018). A metodologia de desenvolvimento de *software* utilizada na Petrobras e apresentada anteriormente foi levada em consideração por ambos autores.

2.5 MÉTODO AUTOMÁTICO DE TESTES DE CONFORMIDADE DE SIS

Tendo em vista o procedimento manual adotado pela Petrobras para a execução do teste de conformidade de CLP realizado durante o TAF e a exploração de um número limitado de condições, Veiga (2018) apresenta um método complementar para diagnóstico de falhas de execução da implementação lógica de SIS.

A estratégia adotada não leva em consideração as informações a respeito da estrutura lógica da implementação, a geração do teste é realizada a partir da especificação do sistema.

Figura 5 – Ilustração do método de geração e execução de testes proposto por Veiga, 2018.



Fonte: (VEIGA, 2018).

Conforme apresentado na Figura 5, o método utiliza a Matriz de Causa e Efeito, documento regido por norma utilizado para expressar as especificações do sistema, para a geração de casos de teste (ramo da esquerda) e para a geração de modelos de Oráculos (ramo da direita). Estes Oráculos expressam funções de segurança específicas dos SIS e são utilizados para o diagnóstico de falhas durante a execução dos casos de teste.

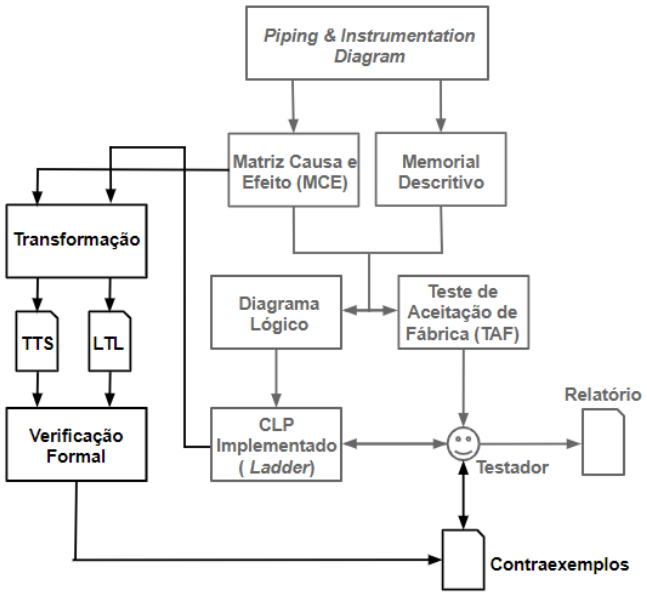
Os casos de teste são organizados em uma Lista de Casos de Teste que engloba as condições que serão testadas para análise de todas as funções de segurança. Estes casos de teste são executados ordenadamente definindo valores de entrada do CLP. As saídas geradas pelo CLP são por fim avaliadas pelo oráculo para o diagnóstico de inconformidades do sistema frente às especificações descritas na Matriz de Causa e Efeito. Um relatório informa as inconformidades encontradas.

No trabalho é apresentada uma ferramenta experimental que engloba edição de MCE, geração de casos de teste, execução do teste no CLP e a apresentação das inconformidades encontradas. A ferramenta foi aplicada para o teste do CLP responsável pelo sistema de Fogo e Gás de uma plataforma exploratória de petróleo. Apresentamos a seguir a alternativa de verificação formal proposta por Dos Reis, 2018.

2.6 MÉTODO AUTOMATIZÁVEL PARA VERIFICAÇÃO FORMAL DE SIS

Tendo em vista que os métodos formais utilizam proposições lógicas temporais, pouco usuais para engenheiros, Dos Reis (2018) propôs a utilização da linguagem intermediária de alto nível FIACRE juntamente com a metodologia atual de desenvolvimento da Petrobras apresentada na Figura 2. O autor propõe adicionar uma cadeia de verificação formal, conforme ilustrado na Figura 6.

Figura 6 –Metodologia de Desenvolvimento de SIS com o uso de Verificação Formal.



Fonte: Adaptado de (DOS REIS, 2018).

As especificações documentadas na forma de MCE são, sistematicamente, traduzidas para fórmulas de Lógica Temporal Linear (LTL). Em seguida, é realizada a tradução do *software* de CLP implementado em Diagrama *Ladder* para um modelo formal expresso em FIACRE. Este modelo é então compilado para o Sistema de Transição Temporizada (*Timed Transition System* – TTS) pela ferramenta FRAC e a verificação formal deste modelo é feita através de *software open source* que recebe as propriedades de segurança expressas em LTL. Eventuais contraexemplos encontrados pelo verificador serão testados no CLP.

2.7 CONSIDERAÇÕES FINAIS DA SEÇÃO

Valendo-se dos fatos de que o procedimento de *model checking* é aplicado sobre um modelo, uma abstração de um sistema, e o teste exaustivo pode não ser realizável por limitações computacionais devido à complexidade combinatória, podemos inferir que os métodos de teste de conformidade e de *model checking* apresentam vantagens e limitações complementares. Por este motivo, pretende-se utilizar a técnica de Teste Baseado em Aprendizagem para verificar a conformidade de sistemas.

A técnica Teste Baseado em Aprendizagem combina o algoritmo de *model checking* com a inferência de modelos e o SUT de maneira iterativa (MEINKE; NIU; SINDHU, 2012). O presente trabalho é um estudo da aplicabilidade desta técnica para SIS. Apresentamos mais detalhadamente esta técnica na próxima seção.

3 TESTE BASEADO EM APRENDIZAGEM

O Teste Baseado em Aprendizagem (*Learning based Test* - LBT) é um método que, a partir dos requisitos funcionais do sistema, realiza algoritmicamente testes de conformidade, os quais são orientados por um modelo comportamental do SUT. Este modelo é gerado por um algoritmo de aprendizado e atualizado à medida que novos testes são realizados. Desta forma, o método realiza a geração e execução de casos de teste e a formulação do veredicto (MEINKE; NIU; SINDHU, 2012).

3.1 PROCEDIMENTO LBT

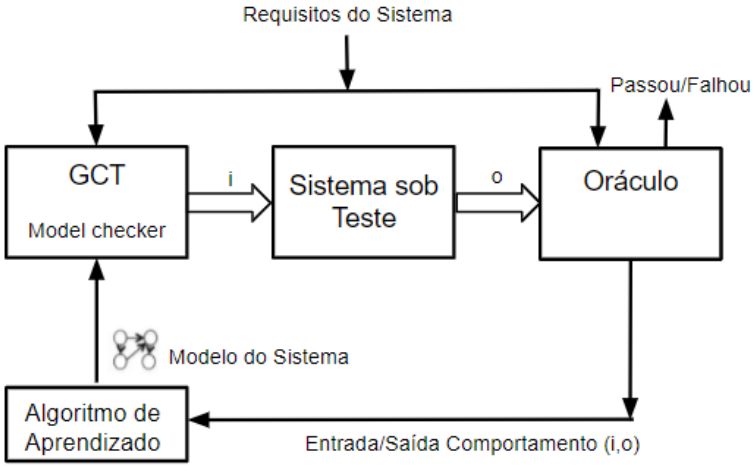
A Figura 7 apresenta a arquitetura simplificada do método de Teste Baseado em Aprendizagem e seus quatro principais componentes: *Gerador de Casos de Teste* (GCT), *Sistema sob Teste*, *Oráculo* e *Algoritmo de Aprendizado*. Um algoritmo de *model cheking* é utilizado como GCT, por meio dos contraexemplos encontrados. O Oráculo é um procedimento de comparação entre os comportamentos do modelo e do Sistema sob Teste. Esta comparação é realizada quando são testados os contraexemplos do GCT. Comportamentos idênticos, entre modelo e Sistema sob Teste, retratam inconformidade do sistema. Comportamentos distintos, retratam um comportamento do sistema ainda não modelado. O Algoritmo de Aprendizado gera um conjunto de testes que são executados no Sistema sob Teste. As saídas do sistema são utilizadas pelo Algoritmo de Aprendizado na construção de um modelo que retrata o comportamento observado.

Os Requisitos do Sistema, expressos em Lógica Temporal Linear (LTL), são utilizados, primeiramente, pelo *model checker* para verificar o modelo construído. Em caso de inconformidade, a verificação apresenta um contraexemplo que será executado como próximo teste (i). Caso contrário, o próximo teste será definido aleatoriamente. Se *i* é proveniente do *model checker*, o Oráculo irá atuar a fim de identificar se a inconformidade é apenas do modelo ou se está presente no Sistema sob Teste. Este veredicto é construído da seguinte maneira: "SUT passou no teste" caso o sinal de saída (o) observado cumprir os Requisitos do Sistema, caso contrário o "SUT falhou no teste". Desta forma o algoritmo prossegue iterativamente até encontrar um teste negativo validado pelo oráculo ou atingir um critério de parada (por exemplo, tempo de execução ou número máximo de casos de teste).

Apresentamos a seguir um pseudoalgoritmo que implementa o funcionamento do método LBT.

O LBT, conforme apresentado na Figura 7, introduz uma realimentação no processo básico de teste através do uso de um algoritmo de aprendizado, o qual tenta inferir um modelo do SUT com base em todos os dados de teste disponíveis (entradas e saídas). Este modelo pode então ser automaticamente analisado (via procedimento de *model checking*) para tentar identificar traços do modelo que não satisfazem a uma dada especificação. Quaisquer eventuais traços de contraexemplos serão aplicados como um novo caso de teste. Estes novos casos de teste, quando executados, vão provendo informações adicionais acerca do comportamento do SUT, fazendo com que o algoritmo produza uma série de modelos convergentes com comportamento do sistema, conforme explicamos algoritmicamente a seguir.

Figura 7 - Arquitetura LBT: Gerador de Casos de Teste, Sistema sob Teste, Oráculo e Algoritmo de Aprendizado.



Fonte: Adaptado de (MEINKE; NIU; SINDHU, 2012).

Algoritmo 1 - Pseudoalgoritmo Teste Baseado em Aprendizagem.

```
1. LBT (SUT: sistema sob teste, i_0: conjunto inicial de testes, requisitos:
especificação funcional em lógica temporal)
2. I <- nova Pilha([i_0]);
3. M <-  novo Modelo( );
4. Erro <- não encontrado;
5. Enquanto (Erro não encontrado e critério de parada não atingido)
6. Entrada <- I.pop( );
7.Saída_SUT <- SUT.executa(Entrada);
8.M <- M.aprende(Entrada,Saída);
9.Contraexemplo<- M.model_check(requisitos);
10.Se (Contraexemplo não vazio) //insere novo teste na pilha
11.I.push(contraexemplo);
12.Entrada <- I.pop( );
13.Saída_SUT <- SUT.executa(Entrada);
14.Saída_Modelo <- SUT.modelo(Entrada);
15.Se Saída_SUT = Saída_Modelo
16.Erro <- encontrado;
17.Senão //gera novo teste aleatório
18.I.push(teste aleatório)
```

Fonte: Elaboração do autor (2020).

O Algoritmo 1 tem acesso às entradas e saídas do SUT, recebe um conjunto inicial de casos de teste (i_0), e os requisitos funcionais do sistema (requisitos) expressos em lógica temporal na linha 1. Entre as linhas 5 e 18, o procedimento LBT se desenvolve realizando, iterativamente, testes e refinamentos no modelo até que um erro seja encontrado, todos os requisitos sejam verificados (linha 5) ou algum critério de parada seja atingido (linha 6). Na linha 8 o próximo teste é executado no SUT. O par (Entrada, Saída) é utilizado pelo algoritmo de aprendizado no refinamento/construção do modelo (M) na linha 8. Esse modelo é verificado frente aos requisitos do sistema pelo *model checker*, na linha 9. Quando o modelo não satisfaz os requisitos, um contraexemplo é fornecido pelo *model checker* (linha 12). Esse contraexemplo servirá de entrada para o SUT e também para o modelo possibilitando ao Oráculo identificar se o erro é de fato uma falha do sistema (linhas 15 e 16). Quando o *model checker* não encontrar um contraexemplo (linha 17) a nova entrada do SUT será fornecida por um gerador aleatório (linha 18). O algoritmo prossegue, iterativamente, aplicando testes, gerando novos modelos e verificando-os até que uma falha seja encontrada (linha 16), ou que seja atingido algum critério

de parada. Possíveis critérios incluem máximo tempo de execução ou um número máximo de iterações.

O LBT, por adicionar a uma abordagem baseada em testes um algoritmo de *model checking,* possui algumas peculiaridades quando comparado aos dois métodos apresentados no capítulo anterior. O LBT substitui a dificuldade de encontrar diretamente um modelo, pela dificuldade de encontrar um conjunto de entradas/saídas que consigam representar o comportamento da planta. Isso, conforme descrito anteriormente no algoritmo de aprendizado, pode não convergir para um nível de abstração que possibilite a identificação de uma falha nem a comprovação do cumprimento de todos os requisitos, encerrando sua execução sem veredicto. O algoritmo de *model checking* é utilizado na identificação de divergências entre modelo e sistema, gerando, a partir destas, casos de teste. É de grande valia lembrar também, que seu veredicto é baseado no modelo, uma abstração que não preserva todas as características do sistema, o que é mais uma limitação deste método.

3.2 ALGORITMOS DE APRENDIZADO

No contexto de projeto de sistemas ditos inteligentes, os quais podem interagir e aprender com estas interações, um relevante problema com diferentes aplicações é a geração automática de modelos a partir de dados (GIANTAMIDIS; TRIPAKIS, 2016). Algumas aplicações incluem: *chatbots*, programas de computador que tentam simular um ser humano em conversas com pessoas; análise de relatórios empregados no conceito da Indústria 4.0, os quais utilizam dados estatísticos dos processos para orientar melhorias; o emprego em validação de *softwares* através de modelagem funcional.

Na arquitetura LBT, o algoritmo de aprendizado tem o papel de traduzir as informações provenientes dos testes realizados no sistema para um modelo, que será utilizado para prover novos casos de teste através do algoritmo de *model checking* (ver Seção 2.4). Suas variantes são numerosas, desde o conteúdo das informações, sua sintaxe, até a linguagem de modelagem. Os algoritmos de aprendizado na área de autômatos podem ser classificados quanto ao modo de fornecimento das informações, em aprendizado *online* ou *offline*. Procedimentos de aprendizado *offline* possuem um conjunto pré-determinado e fixo de exemplos que subsidiarão a construção do modelo. Ao passo que, no aprendizado *online*, novos exemplos podem ser iterativamente solicitados pelo próprio algoritmo durante a execução.

Os algoritmos podem ser categorizados também em *incremental*, quando uma sequência de autômatos hipóteses são construídos até convergirem ao modelo final ou em *não-incremental*, quando apenas um modelo é elaborado ao longo de todo o processo de aprendizado.

A seguir, apresentamos os três algoritmos elencados para o estudo. Iniciando pelo algoritmo L* (ANGLUIN, 1987), o qual foi o precursor do aprendizado *online*. Escolhido por sua baixa complexidade computacional, sua proposta consiste em um procedimento para inferência de autômatos determinísticos de estados finitos (ADEF) a partir de exemplos de cadeias de eventos aceitas ou não pela linguagem em aprendizado. O segundo algoritmo que abordamos nesta seção foi apresentado em (MEINKE; SINDHU, 2012) e desenvolvido para Teste baseado em Aprendizagem. Ele utiliza o conceito de distinção de sequências para, de maneira *online* e *incremental*, aprender um ADEF. Por fim, apresentamos uma proposta recente que aprende de maneira *offline* e *não-incremental* uma máquina de Moore (GIANTAMIDIS; TRIPAKIS, 2016). Este, por sua vez, modela as informações do sistema e vai, iterativamente, fundindo alguns estados específicos até chegar a um modelo final.

Contudo, para facilitar o entendimento desta seção, iniciamos apresentando alguns conceitos e notações que serão utilizados na descrição dos algoritmos.

3.2.1 Notações e Conceitos Preliminares

Seja Σ um conjunto finito de símbolos, utilizamos Σ^* para denotar o conjunto de todas as palavras finitas sobre Σ incluindo a palavra vazia λ. Para palavras $\alpha, \beta \in \Sigma^*$, $\alpha\beta$ denota sua concatenação. Para $\alpha, \beta, \gamma \in \Sigma^*$, se $\alpha = \beta\gamma$ então β é chamado de *prefixo* de α e γ é chamado *sufixo* de α. *Prefixo-fechamento* de α é o conjunto de todos os *prefixos* de α. *Sufixo-fechamento* é definido de forma análoga (CARMEL; MARKOVITCH, 1996).

Um autômato determinístico de estados finitos (ADEF) é uma quíntupla $A = (\Sigma, Q, F, q_0, \delta)$, sendo:

- Σ um conjunto finito de símbolos (eventos) que definem o alfabeto;
- Q um conjunto finito de estados do autômato;
- $F \subseteq Q$ um conjunto de estados finais (marcados);
- $q_0 \in Q$ um estado inicial do autômato;
- $\delta: Q \times \Sigma \rightarrow Q$ uma função de transição, possivelmente parcial.

Uma função de transição $\delta(q_i, b) = q_j$ denota que, quando no estado $q_i \in Q$ com a ocorrência do evento 'b', o autômato irá para o estado $q_j \in Q$ em um passo. Estendemos a função $\delta^*: Q \times \Sigma^* \to Q$, definida por $\delta = (q, \lambda) = q$ e $\delta^* = (q, b_1, \dots, b_n) = \delta(\delta^*(q, b_1, \dots, b_{n-1}), b_n)$. Quando a função de transição em um ADEF é parcial, dizemos que o ADEF é incompleto.

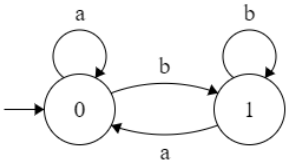
Uma linguagem definida sobre um conjunto de eventos E é um conjunto de palavras de comprimento finito formadas a partir de eventos em E. Um autômato é um dispositivo capaz de representar uma linguagem de acordo com regras bem definidas. Uma linguagem é dita regular se ela pode ser marcada por um ADEF. A linguagem $L(A)$ aceita por A é o conjunto de todas as palavras $\alpha \in \Sigma^*$ tal que $\delta^*(q_0, \alpha) \in F$.

Uma máquina de Moore é uma tupla $A = (I, O, Q, F, q_0, \delta)$, sendo:

- I um conjunto finito de símbolos de entrada;
- O um conjunto finito de símbolos de saída;
- Q um conjunto finito de estados do autômato;
- $F \subseteq Q$ um conjunto de estados finais (marcados);
- $q_0 \in Q$ um estado inicial do autômato;
- $\delta: Q \times \Sigma \to Q$ uma função de transição completa;
- $Y: Q \times \mathrm{I} \to O$ uma função de saída completa.

A Figura 8 exemplifica uma Máquina de Moore, em que o estado inicial apresenta o símbolo 0 como saída. O segundo estado, que é alcançado a partir do inicial pela ocorrência do símbolo b, apresenta 1 como símbolo de saída.

Figura 8 – Máquina de Moore no conjunto de símbolos de entrada e saída, $I = \{a, b\}$ e $O = \{0,1\}$, respectivamente.



Fonte: Elaboração do autor (2020).

A sequência de símbolos de entrada $abba$, para esta Máquina de Moore, gera como saída 00110. O primeiro símbolo da sequência de saída gerada é a saída apresentada pelo estado inicial, do segundo símbolo em diante são apresentados os símbolos dos respectivos estados

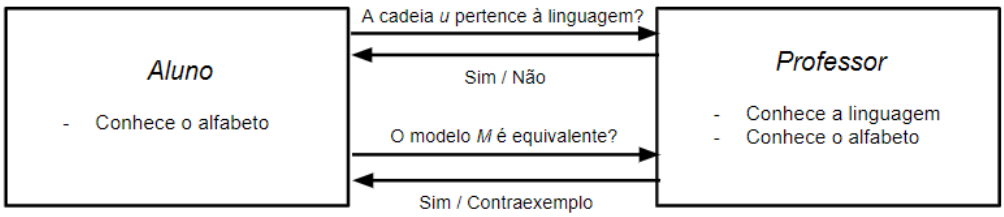
alcançados pela ocorrência dos eventos de entrada. Desta forma, o evento a mantém a máquina de Moore no estado inicial, resultando no segundo 0 da sequência de saída.

3.2.2 Algoritmo L*

O algoritmo L* foi apresentado por (ANGLUIN, 1987) como solução para o problema da identificação de linguagens regulares desconhecidas, a partir de exemplos positivos e negativos, sendo, respectivamente, as cadeias de eventos pertencentes e não pertencentes ao conjunto regular. A proposta introduziu o conceito de aprendizado de ADEF através de perguntas de um *aluno* para um *professor*.

A Figura 9 ilustra a interação entre o *aluno* e o *professor*. O *professor* conhece uma linguagem regular fixa L sobre um alfabeto conhecido. O *aluno* conhece o alfabeto e deseja construir um ADEF aceitador de L. Para isto, o *aluno* vai, iterativamente, perguntando na forma "esta cadeia pertence à linguagem?", até o momento em que possua informações suficientes para a construção do seu ADEF (modelo) hipótese.

Figura 9 – Estrutura iterativa no algoritmo L*.

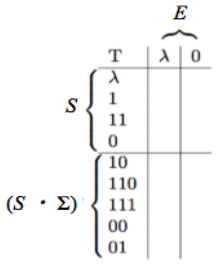


Fonte: Elaboração do autor (2020).

Uma vez obtido o autômato hipótese, o *aluno* realiza outro tipo de pergunta "O modelo M é equivalente?", como apresentado na Figura 9. Momento em que o *professor* verifica sua congruência frente à linguagem regular. Caso esteja correto, o algoritmo apresenta o modelo e encerra a execução. No entanto, se não estiver correto, o *professor* fornecerá um contraexemplo, uma cadeia de eventos que pertence à linguagem, mas não é aceita pelo modelo. Esta informação será utilizada no refinamento do modelo-hipótese de maneira incremental, juntamente às demais respostas obtidas previamente, até que um próximo autômato seja formulado. O algoritmo segue indefinidamente até que a hipótese seja aprovada na verificação.

Toda informação coletada no processo de aprendizado é organizada em uma Tabela de Observação. Quando esta tabela contiver informação suficiente para a construção de um autômato determinístico, o algoritmo L* utiliza a tabela para construir um ADEF.

Figura 10 - Tabela de Observação.



Fonte: Elaboração do autor (2020).

A Tabela de Observação (Figura 10) é constituída por uma tupla (S, E, T) composta por um conjunto S, finito, não-vazio e *prefixo-fechado* de palavras em Σ^*; um conjunto E finito, não-vazio e *sufixo-fechado* de palavras em Σ^*; e uma função T finita de mapeamento $((S \cup S \cdot \Sigma) \cdot E)$ para $\{0,1\}$. A interpretação de T é que $T(u)$ é 1 se, e somente se, u é um membro do conjunto desconhecido U.

Durante o preenchimento da Tabela de Observação, (S, E, T), através das perguntas realizadas ao *professor* duas propriedades são verificadas a fim de garantir que as informações coletadas são suficientes para conjeturar um autômato determinístico finito. A primeira propriedade é a consistência. A tupla é consistente se para todo par de elementos em S com mesma sequência E na tabela, todas extensões com o mesmo símbolo de Σ também possuem mesmo E. A segunda propriedade de (S, E, T) é fechamento. Este é o caso em que para cada t em $S \cdot \Sigma$ existe um s em S tal que $linha(t) = linha(s)$. Em que $linha(x)$ é a função que retorna uma tupla com todos os elementos da linha x em E da Tabela de Observação.

O algoritmo L* inicia a partir de uma tabela com $S = E = \{\lambda\}$, ela é aumentada à medida em que o algoritmo é executado. Um pseudoalgoritmo é apresentado no Algoritmo 2.

Algoritmo 2 - Pseudoalgoritmo L*.

```
1. Define: S = {λ}, E = {λ}
2. Pergunta para o professor: "λ pertence a U? a pertence a U?" Para todo a ∈ Σ.
3. Constrói a Tabela de Observação inicial (S,E,T).
4. Enquanto (S,E,T) é não fechada e não consistente:
5. Se (S,E,T) é não consistente:
6.Então: encontra s1 e s2 em S, a ∈ Σ, e e ∈ E tal que
7.linha(s1)=linha(s2) e T(s1 · a · e) ≠ T(s2 · a · e),
8.Adiciona a · e em E,
9.Extende T para ((S ∪ S · Σ) · E) através das perguntas: pertence a U?
10.Se (S,E,T) é não fechada,
11.Então: encontra s1 ∈ S e a ∈ Σ tal que linha(s1 · a) é diferente de linha(s)
para todo s ∈ S,
12.Adiciona s1 · a em S,
13.Extende T para ((S ∪ S · Σ) · E) através das perguntas: pertence a U?
14.(S,E,T) é fechada e consistente, fazemos M = M(S, E, T).
15.Conjectura M.
16.Se o Professor responder com um contraexemplo t:
17.Então: Adiciona t e todos seus prefixos em S
18.Extende T para ((S ∪ S · Σ) · E) através das perguntas: pertence a U?
19.Até o Professor responder "sim" para a conjectura M.
20.Termina e apresenta M.
```

Fonte: Adaptado de (ANGLUIN, 1987).

Analisemos as propriedades apresentadas anteriormente para a Tabela 1. Quando estas são verdadeiras, ou seja (S, E, T) fechada e consistente, podemos definir um modelo correspondente M sob o alfabeto Σ como:

- $Q = \{linha(s): s \in S\}$;
- $q_0 = linha(\lambda)$;
- $F = \{linha(s): s \in S \ \wedge T(s) = 1$;
- $\delta(linha(s), a) = linha(s \cdot a)$.

Tabela 1 – Exemplo de Tabela de Observação: $T = \{\lambda, a, b\}$, $E = \{\lambda\}$.

T	λ
λ	0
a	1
b	1
aa	0
ab	1
bb	0

Fonte: Elaboração do autor (2020).

A Tabela 1 é inconsistente, uma vez que a e b possuem a mesma sequência E (representando um mesmo estado do ADEF), no entanto quando concatenados com o evento b, resultando nas cadeias ab e bb (duas últimas linhas da tabela) seus conteúdos em E se diferem (representando estados distintos no ADEF). A Tabela 1 é completa pelo fato de todos os conteúdos de E estarem representados na parte superior da tabela (T).

A seguir apresentamos um exemplo de execução do algoritmo L* para um caso simplificado. Este caso será utilizado para elucidar a execução dos demais algoritmos de aprendizado de autômatos apresentados neste capítulo.

Caso Simplificado: Aprender o conjunto regular *U* das palavras que possuem número par de *1*s e não são vazias sobre o alfabeto $\Sigma = \{0,1\}$.

Execução Algorítmica: Inicialmente, L* pergunta: "λ, *0* e *1* pertencem a *U*?". A Tabela de Observação inicial T_1 é apresentada na Tabela 2.

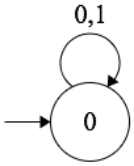
Tabela 2 - Tabela de Observação T_1: $S = E = \{\lambda\}$.

T	λ
λ	0
0	0
1	0

Fonte: Elaboração do autor (2020).

Esta tabela é consistente e fechada, então L* pode conjecturar o aceitador M_1 apresentado na Figura 11.

Figura 11 - Aceitador M_1.



Fonte: Elaboração do autor (2020).

O estado inicial de M_1 é q_0. M_1 não é um aceitador correto para U, então o *professor* seleciona um contraexemplo. Neste caso, expomos que o contraexemplo 11 é selecionado, este pertencente a U mas rejeitado por M_1.

Tabela 3 - Tabela de Observação T_2: $S = \{\lambda, 1, 11\}$; $E = \{\lambda\}$.

T	λ
λ	0
1	0
11	1
0	0
10	0
110	1
111	0

Fonte: Elaboração do autor (2020).

Para processar o contraexemplo 11, L* adiciona as palavras 1 e 11 em S (a palavra λ já está em S), e pergunta: "*10*, *11*, *110* e *111* pertencem a *U*?" para construir a Tabela 3.

A Tabela 3, no entanto, é inconsistente, pois $linha(\lambda) = linha(1)$, mas $linha(1) \neq linha(11)$. Então L* adiciona a palavra 1 em E e pergunta: "*01*, *101*, *1101* e *1110* pertencem a *U*?" para construir a Tabela 4.

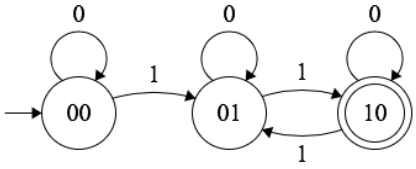
Tabela 4 - Tabela de Observação T_3: $S = \{\lambda, 1, 11\}$; $E = \{\lambda, 1\}$.

T	λ	1
λ	0	0
1	0	1
11	1	0
0	0	0
10	0	1
110	1	0
111	0	1

Fonte: Elaboração do autor (2020).

Essa tabela é consistente e fechada, então L* pode conjecturar o aceitador M_2 apresentado na Figura 12.

Figura 12 - Aceitador M_2.



Fonte: Elaboração do autor (2020).

M_2 é um correto aceitador para a linguagem U, então o *professor* responde a esta conjectura com "sim, é equivalente" e L* encerra sua execução apresentando M_2 como o modelo que produz o conjunto regular das palavras não vazias que possuem número par de *1*s.

3.2.3 Distinção Incremental de Sequências

O algoritmo de Distinção Incremental de Sequências (IDS - *Incremental Distinguishing Sequences*) foi apresentado em (MEINKE; SINDHU, 2012) como uma alternativa ao aprendizado incremental de ADEFs para aplicações em teste formal de software e inferência de modelos. Para isto, o procedimento recebe como entrada um conjunto S formado por cadeias de eventos. O algoritmo estrutura-se na forma *aluno* e *professor*, semelhante ao algoritmo L* (ver Seção 3.2.2). No entanto, a interação destes personagens no algoritmo IDS se dá entorno do único tipo de pergunta que é realizada pelo *aluno*: "a cadeia pertence à linguagem? ".

O artifício utilizado pelo algoritmo para definir as cadeias que serão levadas ao *professor* se dá através da construção iterativa de dois conjuntos específicos: um conjunto T de palavras, o qual é derivado do conjunto de entrada S; e um conjunto V de sufixos. As cadeias que serão questionadas são definidas pela concatenação de todas as palavras com todos os sufixos, ou seja, todos os elementos do conjunto T concatenados a todos elementos de V. Obtidas as respostas na forma, "pertence" ou "não pertence à linguagem", o algoritmo vai agrupando as palavras de acordo com os sufixos que as fazem pertencer à linguagem regular em aprendizado.

Por exemplo, dados $T = \{ca, abc, bba\}$ e $V = \{b, aa\}$ as cadeias questionadas pelo *aluno* ao *professor* seriam: cab, $caaa$, $abcb$, $abcaa$, $bbab$ e $bbaaa$. Caso apenas as cadeias cab, $caaa$ e $abcb$ pertençam à linguagem regular em estudo, o algoritmo organizará este conjunto de informações conforme apresentado na Tabela 5.

Tabela 5 – Exemplo de Tabela de Informações organizadas pelo algoritmo IDS.

$E(ca) = \{b, aa\}$
$E(abc) = \{b\}$
$E(bba) = \emptyset$

Fonte: Elaboração do autor (2020).

Na primeira linha da Tabela 5 é definido $E(ca)$, o conjunto dos sufixos sobre a palavra ca que fazem a cadeia de eventos pertencer à linguagem em aprendizado. Como neste exemplo definimos que as palavras cab e $caaa$ pertencem à linguagem, os sufixos b e aa pertencem ao conjunto $E(ca)$. Para a palavra abc, segunda linha da Tabela 5, o sufixo aa não faz a cadeia pertencer à linguagem, desta forma aa não pertence ao conjunto $E(abc)$. Prosseguindo desta forma os conjuntos $E(\alpha)$ para $\forall \alpha \in$ T são construídos e atualizados a medida que os conjuntos das palavras T e dos sufixos V crescem iterativamente, conforme apresentamos a seguir.

A entrada do algoritmo é um conjunto de palavras de entrada S, que vão sendo processadas uma a uma (α). A partir destas é formado o conjunto de "palavras em estudo" T, em que $T = \{\alpha\} \cup \{\delta(\alpha, \beta) | \forall \beta \in \Sigma\}$. As perguntas realizadas são: "$\alpha\beta$ pertence ao conjunto?", para $\forall \alpha \in$ T, $\forall \beta \in$ V. Quando o *professor* responder "sim", o *aluno* guardará esta informação na forma: $E_k(\alpha) = E_{k-1}(\alpha) \cup \{\beta\}$ denotando que, o sufixo β para a palavra α foi adicionado ao novo conjunto dos sufixos aceitos em U na k-ésima iteração. Caso contrário, $E_k(\alpha) =$

$E_{k-1}(\alpha)$, que representa que o conjunto dos sufixos sob a palavra α que fazem a palavra pertencer ao conjunto regular U segue o mesmo.

O algoritmo IDS inicia sua execução buscando informações a respeito da cadeia vazia λ, a partir dela é definido o primeiro conjunto de palavras em estudo, $T_0 = \{\lambda\} \cup \{\beta | \forall \beta \in \Sigma\}$. O conjunto inicial de sufixos é $V = \{\lambda\}$. As perguntas ao *professor* são realizadas, e as informações obtidas modeladas em M_k para $k = 0$ como:

- $Q = \{E_k(a) | a \in T_k\}$;
- $q_0 = E_k(\lambda)$;
- $F = \{E_k(a) | a \in T_k \wedge \lambda \in E_k(a)\}$.

As transições são definidas como segue:

Algoritmo 3 - Pseudoalgoritmo de Construção do Modelo IDS.

```
1. Para toda palavra (a) executada:
2. Se E_k (a) = ∅:
3.Então: δ(E_k(a),b) <- {E_k(a)|∀b ∈ Σ}
4. Senão: δ (E_k (a),b) <- {E_k(ab)|∀b ∈ Σ}
5. Para toda palavra em estudo, mas não executada (β):
6. Se para toda palavra a executada: {E_k(a)≠E_k(β) e E_k(β)≠∅}
7.Então: δ(E_k(β),b) <- ∅)|∀b ∈ Σ}
```

Fonte: Adaptado de (MEINKE; SINDHU, 2012).

Construído o modelo inicial M_0, o algoritmo busca no conjunto de entrada S seu primeiro elemento (a). A partir dele, T_1 é definido adicionando ao conjunto da iteração anterior (T_0) a cadeia a juntamente com $\{ab | \forall b \in \Sigma\}$. As perguntas referentes a todas as combinações dos conjuntos T_1 e V são realizadas e as informações organizadas conforme apresentado na Tabela 5.

Eventualmente, inconsistências frente às informações obtidas surgirão na forma $E_k(x) = E_k(y)$ e $E_k(xy) \neq E_k(yy)$ com $x, y \in \Sigma^*$, análogo à inconsistência das Tabelas de Observação apresentadas na Seção 3.2.2. Estas são contornadas pelo algoritmo através da adição do sufixo causador da inconsistência y ao conjunto de sufixos V, análogo à coluna adicionada à Tabela de Observação (Seção 3.2.2). Esse novo sufixo será então combinado junto a todo o conjunto T, e novas perguntas são dirigidas ao *professor*.

Com um conjunto de informações consistentes, o algoritmo verifica se o modelo atual é congruente com a palavra α em processamento do conjunto S, ou seja, $E_k(\alpha) = Q$. Caso

afirmativo, o modelo não necessita de refinamento e o algoritmo busca a próxima palavra no conjunto de entrada S. Caso contrário, o modelo será redefinido conforme descrito anteriormente no Algoritmo 3.

Desta forma, a execução prossegue: buscando um a um todos os elementos a de S; incorporando a e $\{ab|\forall b \in \Sigma\}$ ao conjunto T; combinando as palavras com os sufixos; buscando as informações junto ao *professor* por meio das perguntas; agrupando os sufixos que fazem as palavras pertencerem à linguagem em estudo; refinando um modelo na forma de ADEF, até apresentar o modelo final. Um pseudocódigo da rotina IDS é apresentado no Algoritmo 4.

Algoritmo 4 - Pseudoalgoritmo IDS.

```
1. i = 0, k = 0, t = 0, v_i = λ, V = {v_i} //Inicialização
2. P_0 = {λ}, P'_0 = P_0 ∪ {d_0}, T_0 = P_0 ∪ Σ
3. E_0(d_0) = ∅
4. Para todo α ∈ T_0 {
5."Pergunta: α pertence a U?"
6.Se o professor responde "sim"
7.E_0(α) = {λ}
8.Senão
9.E_0(α) = ∅}
10. Refina_Partição(T_0)
11. Constrói a conjectura inicial M_0
12. Enquanto (S≠ vazio) {
13.ler (S, α)
14.k= k+1, t = t+1
15.P_k = P_k-1 ∪ {α}
16.P'_k= P_k ∪ {d_0}
17.T_k = T_k-1 ∪ {α} ∪ {f (α, b) | b ∈ Σ}
18.T'_k= T_k ∪ {d_0}
19.∀α ∈ T_k - T_k=1 {
20.Para j = 0 até j = i {
21."Pergunta: αv_j pertence a U?"
22.Se o professor responde sim
23.E_i(α) = {v_j}
24.j = j+1}}
25.Refina_Partição(T_k)
26.Se α é consistente com M_t-1
27.M_t = M_t-1
```

```
28.Senão
29.constrói M_t}}
30. Define Refina_Partição (T)
31.Enquanto (∃α,β∈ P'_k e b∈ Σ onde E_k(α)=E_k(β) e E_k(αb)≠ E_k(βb))
32.Para y ∈ ((E_k(αb) - E_k(βb)) ∪ (E_k(βb)- E_k(αb)))
33.v_{k+1} = by
34.V = V ∪ v_{i+1}, i = i+1
35.∀α ∈ T_k {
36."Pergunta: αv_i pertence a U?"
37.Se o professor responde sim
38.Então E_i(α) = E_{i-1}(α) ∪ {v_i}
39.Senão:
40.E_i(α) = E_{i-1}(α)}
```

Fonte: Adaptado de (MEINKE; SINDHU, 2012).

Importante salientar que, dado um conjunto de entrada S insuficiente para retratar o conjunto desconhecido U, o algoritmo apresentará um modelo consistente com todo o conjunto T construído, o qual para todas as cadeias de eventos em T o modelo aceita as pertencentes à linguagem e rejeita as demais. No entanto, para cadeias de eventos não pertencentes a T, o mesmo não ocorre.

Descrevemos a execução do procedimento IDS para o caso simplificado que foi apresentado na subseção anterior. Diferentemente do algoritmo L*, além do *professor,* necessitamos fornecer como entrada um conjunto de palavras que serão processadas pelo algoritmo. Utilizamos o conjunto $S = \{1,110\}$.

A execução inicia-se pelo processamento da palavra λ e sufixo λ, por meio da pergunta: "$\lambda.\lambda$, $0.\lambda$ e $1.\lambda$ pertencem à linguagem?". Com as respostas fornecidas pelo *professor*, construímos a tabela inicial de partições, representando que nenhuma das palavras λ, 0 e 1 seguidos do sufixo pertence ao conjunto das palavras com número par de *1*s, conforme apresentado na Tabela 6.

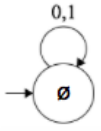
Tabela 6 - Informações Iteração 1.

$E_0(d_0) = \emptyset$
$E_0(\lambda) = \emptyset$
$E_0(0) = \emptyset$
$E_0(1) = \emptyset$

Fonte: Elaboração do autor (2020).

As informações contidas na tabela são consistentes, logo o algoritmo elabora a conjectura inicial M_0 (Figura 13).

Figura 13 - Modelo M_0.



Fonte: Elaboração do autor (2020).

O primeiro elemento de S é agora processado. Perguntando ao *professor*: "*10*.λ e *11*.λ pertencem ao conjunto?", uma vez que já se tem a informação da palavra 1λ. As informações obtidas são acrescentadas, conforme mostra a Tabela 7.

Tabela 7 - Informações Iteração 2.

$E_0(d_0) = \emptyset$
$E_0(\lambda) = \emptyset$
$E_0(0) = \emptyset$
$E_0(1) = \emptyset$
$E_0(10) = \emptyset$
$E_0(11) = \{\lambda\}$

Fonte: Elaboração do autor (2020).

A partição é inconsistente, uma vez que $E_0(\lambda) = E_0(1)$, porém $E_0(\lambda 1) \neq E_0(11)$. Esta inconsistência é causada pelo evento 1, logo este fará parte do conjunto de sufixos a serem testados juntamente com o sufixo λ.

Desta forma processamos o sufixo 1 frente a todas palavras, processo chamado de refinamento das partições. Através das perguntas: "λ*.1*, *0.1*, *1.1*, *00.1*, *01.1*, *10.1* e *11.1* pertencem ao conjunto?". Utilizando estas informações para organizar a Tabela 8.

Tabela 8 - Informações Iteração 3.

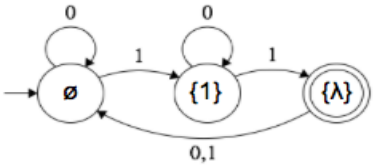
$E_1(\lambda) = \emptyset$
$E_1(0) = \emptyset$
$E_1(1) = \{1\}$
$E_1(00) = \emptyset$
$E_1(01) = \{1\}$

$E_1(10) = \{1\}$
$E_1(11) = \{\lambda\}$

Fonte: Elaboração do autor (2020).

Resolvida a inconsistência, o algoritmo verifica a consistência da palavra em processamento, 1, com o modelo. A cadeia 1 deveria conduzir o autômato ao estado {1}, conforme $E_1(1) = \{1\}$. O estado sequer existe no modelo atual, então o algoritmo necessita refinar a conjectura. O novo autômato M_1 de estado inicial ∅ e estado final $\{\lambda\}$, constituído também pelo estado {1} é apresentado na Figura 14.

Figura 14 - Modelo M_1.



Fonte: Elaboração do autor (2020).

Por fim, a palavra 110 do conjunto de entrada é executada com o algoritmo realizando a pergunta: "*110*.λ, *110*.1, *1100*.λ, *1100*.1, *1101*.λ e *1101.1* pertencem ao conjunto?" ao *professor*. As informações provenientes do *professor* formularam a Tabela 9.

Tabela 9 - Informações Iteração 4.

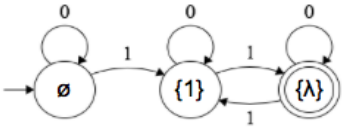
$E_1(\lambda) = \emptyset$
$E_1(0) = \emptyset$
$E_1(1) = \{1\}$
$E_1(00) = \emptyset$
$E_1(01) = \{1\}$
$E_1(10) = \{1\}$
$E_1(11) = \{\lambda\}$
$E_1(110) = \{\lambda\}$
$E_1(1100) = \{\lambda\}$
$E_1(1101) = \{1\}$

Fonte: Elaboração do autor (2020).

As informações contidas na tabela são consistentes. No entanto, a partição referente à palavra 110 é inconsistente com o Modelo M_1. Por este motivo, um novo refinamento na

conjectura é necessário. O modelo obtido a partir das informações contidas na tabela atual possui os mesmos três estados ($\emptyset$, $\{\lambda\}$, $\{1\}$): estado inicial ($\emptyset$) e estado final ($\{\lambda\}$) que a conjectura anterior. No entanto, organiza-se conforme apresentado na Figura 15.

Figura 15 - Modelo M_2.



Fonte: Elaboração do autor (2020).

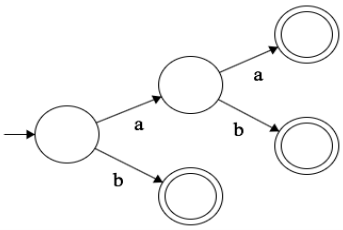
O algoritmo encerra sua execução após ter executado todas as palavras do conjunto de entrada S, apresentando o Modelo M_2 como o modelo aprendido para o conjunto regular U.

3.2.4 Moore MI

O algoritmo Moore MI apresentado em (GIANTAMIDIS; TRIPAKIS, 2016) é uma alternativa para aprendizado de uma máquina de Moore a partir de Traços Entrada-Saída (TES). Um TES apresenta um exemplo do comportamento de uma máquina de Moore, na forma: $(x_1x_2 \dots x_n, y_0y_1 \dots y_n)$, em que $x_1x_2 \dots x_n$ é a sequência de entrada, e $y_0y_1 \dots y_n$ a respectiva sequência de saídas geradas pela máquina de Moore, iniciando por y_0, que contém a saída correspondente ao estado inicial. Estes TES podem ser expressos na forma de exemplos entrada-saída como um par de uma sequência finita de símbolos de entrada e um símbolo de saída: $(x_1x_2 \dots x_n, y)$, onde $x_i \in I$, para $i = 1, 2, \dots n$ e $y \in O$. Observando um ADEF como um caso especial de máquina de Moore com alfabeto de saída binário, podemos utilizar o conceito de exemplos entrada-saída na forma de exemplos positivos e negativos, em que, o exemplo entrada-saída $(w, 1)$ representa que a palavra *w* é aceita pelo ADEF, por tanto, um exemplo positivo, e $(w, 0)$ representando que *w* é rejeitada pelo ADEF, um exemplo negativo.

O procedimento Moore MI que descrevemos a seguir, utiliza os exemplos positivos para construir N Árvores Aceitadoras de Prefixos (*Prefix Tree Acceptors* - PTAs), em que N é o número de bits necessários para representar todos os elementos de O da máquina de Moore a ser aprendida. As PTAs são ADEFs incompletos que aceitam todas as palavras contidas em S_+ e rejeitam as demais. Por exemplo, uma PTA para $S_+ = \{b, aa, ab\}$ é apresentada na Figura 16.

Figura 16 - Uma PTA para $S_+ = \{b, aa, ab\}$.



Fonte: Elaboração do autor (2020).

O algoritmo procede com sua execução buscando iterativamente estados das PTAs que possam ser fundidos. A fusão será aceita se, e somente se, puder ocorrer em todas as PTAs de maneira consistente com seus respectivos exemplos negativos. É importante salientar que a fusão só ocorre em todas PTAs ou em nenhuma, de forma que todas serão sempre idênticas em suas estruturas (estados e transições), distintas apenas nas marcações.

O algoritmo marca, em cinza, os estados que já foram processados e farão parte do modelo final. Os estados marcados na cor preta são os sucessores imediatos dos estados cinzas e representam os estados que estão em processamento. Apresentamos no Algoritmo 5 um pseudoalgoritmo que implementa a técnica Moore MI.

Algoritmo 5 - Pseudoalgoritmo MooreMI.

```
1.Define: MooreMI (conjunto_de_traços, alfabeto_de_entrada,alfabeto_
de_saída):
2.(conjunto_de_exemplos_positivos,
3. conjunto_de_exemplos_negativos,
4. bits_de_saída)
5. <- pré-processa_traços (conjunto_de_traços)
6. N <- teto(log2(|ΣO|))
7. lista_DFA <- constrói_PTA (conjunto_de_exemplos_positivos,ΣI,ΣO)
8. cinza <- {qϵ}
9. preto <- {qa para a ∈ ΣI} ∩ lista_DFA[0]. Q
10. Enquanto preto ≠ ∅:
11.q_preto <- pega_próximo(preto)
12.preto <- preto - {q_preto}
13.fusão_aceita <- falso
14.Para q_cinza ∈ cinza:
15.Para i ∈ {0, …, N-1}:
16.nova_lista_DFA[i] <- fusão(lista_DFA[i],q_cinza, q_preto)
17.Se ∀i ∈ {0, …, N-1}: é_consistente (nova_lista_DFA[i],
18.lista_de_exemplos_negativos [i])
19.Então fusão_aceita <- verdade
```

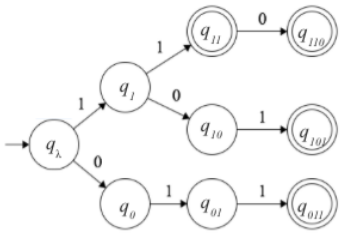
```
20.Se fusão_aceita:
21.Então lista_DFA <- nova_lista_DFA
22.preto <- preto ∪ ({sucessores de uma letra dos estados cinzas} ∩ lista_DFA
[0]. Q)
23.Senão:
24.Então cinza <- cinza ∪ {q_preto}
25.preto <- preto ∪ ({sucessores de uma letra dos estados azuis} ∩ lista_DFA
[0]. Q)
26.Retorna produto(lista_DFA,bits_de_saída).completa_DFA( )
27. Define fusão (DFA, q_cinza, q_preto):
28. q_u <- único_pai_de(q_preto)
29. a_u <- única_entrada_de_para(q_u, q_preto)
30. DFA.δ (q_u, a_u) <- q_cinza
31. Pilha_de_fusão <- [(q_cinza, q_preto)]
32. Enquanto Pilha_de_fusão ≠ [ ]:
33.(q_1, q_2)<- pop (Pilha_de_fusão)
34.Se q_1 = q_2: Então continua
35.Se (q_1, q_2) ≠ (q_cinza,q_preto) e q_2 < q_1:
36.Então q_1, q_2 <- q_2, q_1
37.Então DFA.Q <- DFA.Q - {q_2}
38.Se q_2 ∈ DFA.F:
39.Então DFA.F <- DFA.F ∪{q_1}
40.Para a ∈ DFA.Σ:
41.Se é_definido(DFA.δ(q_2, a)):
42.Se é_definido(DFA.δ(q_1, a)):
43.Então push(fusiona_pilha, DFA.δ(q_1, a), DFA.δ(q_2, a))
44.Senão:
45.Então DFA.δ(q_1, a) <- DFA.δ(q_2, a)
46. Retorna DFA
```

Fonte: Adaptado de (GIANTAMIDIS; TRIPAKIS, 2016).

Descrevemos a execução do algoritmo Moore MI para o caso simplificado o qual foi aplicado os algoritmos anteriormente apresentados nesta seção. Como entrada fornecemos um conjunto com palavras pertencentes à linguagem $S_+ = \{11, 011, 101, 110\}$ e outro com não pertencentes, $S_- = \{\lambda, 0, 1, 10, 111\}$.

O algoritmo inicia sua execução construindo uma PTA a partir de S_+ (Figura 17).

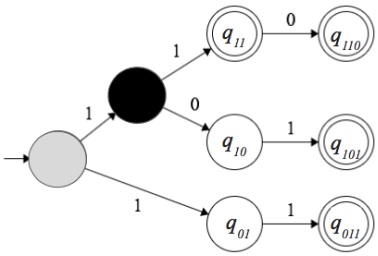
Figura 17 - PTA 1, $S_+ = \{11, 011, 101, 110\}$.



Fonte: Elaboração do autor (2020).

Primeiramente, o estado inicial da PTA passa a ser representado na cor cinza e seus sucessores imediatos, q_0 e q_1, na cor preta. O algoritmo fará uma fusão temporária entre q_λ e q_0, resultando na PTA 2 (Figura 18).

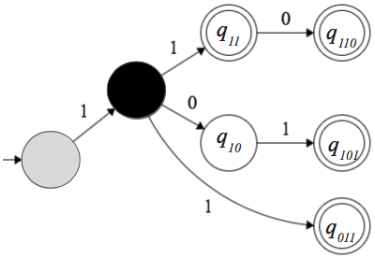
Figura 18 - PTA 2, resultante da fusão de q_λ e q_0.



Fonte: Elaboração do autor (2020).

O evento 1, partindo de q_0, é definido e partindo de q_λ também, logo estes dois estados de destino também passarão por fusão temporária (q_1 e q_{01}), resultando, provisoriamente, na PTA 3 (Figura 19).

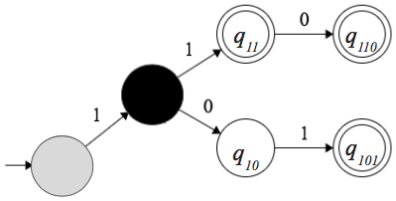
Figura 19 - PTA 3, resultante da fusão de q_1 e q_{01}.



Fonte: Elaboração do autor (2020).

O evento 1 é definido para q_{01} e partindo de q_1 também, logo, estes estados também passarão por fusão temporária (q_{11} e q_{011}), formando a PTA 4 (Figura 20).

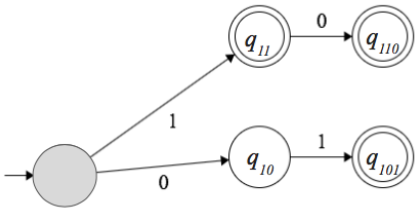
Figura 20 - PTA 4, resultante da fusão de q_{11} e q_{011}.



Fonte: Elaboração do autor (2020).

Nenhum evento a partir do estado q_{011} é definido, então o algoritmo verifica a consistência desta PTA obtida com o conjunto dos exemplos negativos ($S\_$). Como a PTA 4 rejeita todos as palavras pertencentes a S, todas as fusões realizadas são aceitas, o algoritmo fará agora uma fusão temporária entre os estados cinza (q_λ) e preto (q_1), formulando, assim, a PTA 5 (Figura 21).

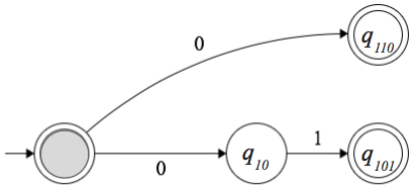
Figura 21 - PTA 5, resultante da fusão de q_λ e q_1.



Fonte: Elaboração do autor (2020).

O evento 0 partindo de q_1 é definido, no entanto para q_λ não. Contudo, o estado resultante seguido do evento 0 levará também ao estado q_{10}. Já o evento 1 é definido para ambos os estados (cinza e preto) e leva a estados distintos, fazendo com que a fusão destes (q_1 e q_{11}) seja realizada também provisoriamente. Como q_{11} é estado final, o estado fusionado também será, resultando na PTA 6 (Figura 22).

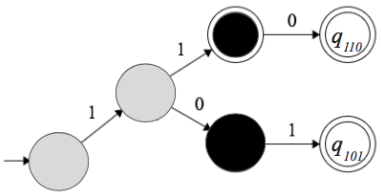
Figura 22 - PTA 6, resultante da fusão de q_1 e q_{11}.



Fonte: Elaboração do autor (2020).

Como nenhum evento é definido a partir do estado q_{110}, o processo de fusão provisória foi concluído. O algoritmo verifica a consistência da PTA obtida com o conjunto S_{-}. A inconsistência pode ser verificada logo no estado inicial, o qual aceita a palavra λ. As fusões provisórias são todas refutadas, momento em que o algoritmo retorna à PTA 4 tornando o estado preto cinza e marcando seus sucessores na cor preta, conforme ilustrado na Figura 23.

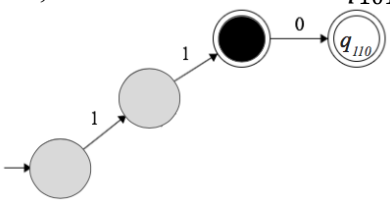
Figura 23 - PTA 7, fusões (q_{11} e q_{011}) (q_1 e q_{11}) refutadas.



Fonte: Elaboração do autor (2020).

O algoritmo prossegue sua execução na tentativa de fundir o estado preto (q_{10}) do ramo inferior com seu antecessor imediato cinza (q_1). O evento 1 a partir de q_{10} é definido, e partindo de q_1 também. Os estados de destino, respectivamente q_{101} e q_{11}, são fusionados provisoriamente, formando a PTA 8 (Figura 24). A consistência deste modelo com o conjunto S_{-} ratifica as fusões provisórias.

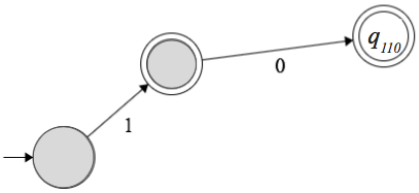
Figura 24 - PTA 8, resultante da fusão de q_{101} e q_{11}.



Fonte: Elaboração do autor (2020).

A fusão do estado preto com seu predecessor imediato é agora processada. Como apenas o evento 0 é definido a partir do estado preto, e este estado é marcado, obtemos a PTA 9 (Figura 25).

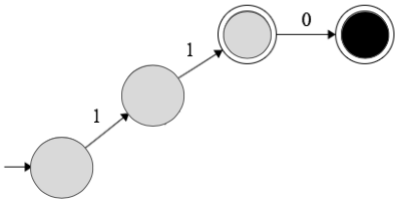
Figura 25 - PTA 9, resultante de fusão dos estados preto e cinza.



Fonte: Elaboração do autor (2020).

O estado q_{110} não possui nenhum sucessor, logo a consistência com o conjunto S_- é verificada. A aceitação da palavra 1 faz com que a última fusão provisória realizada seja rejeitada, marcando o estado preto da PTA 8 como cinza e seu sucessor como preto, resultando na PTA 10 (Figura 26).

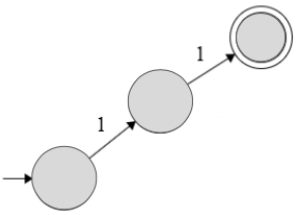
Figura 26 - PTA 10, fusão dos estados preto e cinza refutada.



Fonte: Elaboração do autor (2020).

O presente modelo nos mostra que resta processar a última possível fusão, entre os dois últimos estados da PTA 10 (q_{11} e q_{110}). Como o estado preto não possui eventos definidos a partir dele, a fusão é realizada e a consistência com S_- é verificada.

Figura 27 - PTA 11, final.

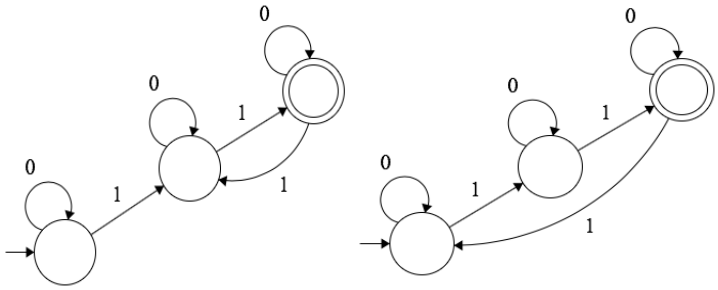


Fonte: Elaboração do autor (2020).

A PTA 11 (Figura 27) é consistente, e como todos estados estão cinzas, ou seja, foram processados, o algoritmo encerra sua etapa de fusões e completa o modelo atentando aos conjuntos de exemplos positivos e negativos que possui.

A partir de $S_+ = \{11, 011, 101, 110\}$, intuitivamente, definem-se os autolaços de 0 nos três estados. No entanto, é indefinido o evento 1 partindo do estado q_{11}, mesmo analisando $S_- = \{\lambda, 0, 1, 10, 111\}$. Sendo os modelos possíveis M_1 e M_2 apresentados na Figura 28.

Figura 28 - Modelos consistentes com S_- e S_+: M_a (esquerda) M_b (direita).



Fonte: Elaboração do autor (2020).

Para resolver esta indeterminação, é necessário prover um conjunto maior de informações sobre o conjunto regular a ser aprendido. Adicionando, por exemplo, 11111 ao conjunto dos exemplos negativos possibilitamos a convergência do algoritmo para o modelo M_a.

3.2.5 Análise dos Algoritmos de Aprendizado

Nesta seção, compararemos o desempenho dos algoritmos para o problema ilustrativo de identificação de um modelo que aceita todas as palavras não vazias com número par de *1*s. Algumas métricas são apresentadas na Tabela 10.

Tabela 10 - Resultados Práticos dos Algoritmos de Aprendizado.

Métrica\Algoritmo	L*	IDS	MooreMI
Número de Modelos construídos	2	2	1
Número de Perguntas realizadas	11	18	-
Número de Iterações	3	4	5
Informação Inicial	-	{λ,0,1}	{λ,0,1,10,11,011,101,110,111}

Fonte: Elaboração do autor (2020).

Tabela 11 - Informações Coletadas.

Algoritmo	Cadeias de Eventos Estudadas
L*	{λ, 0, 1, 00, 01, 11, 101, 110, 111, 1101, **1110**}
IDS	{λ, 0, 1, 10, 11, 001, 011, 101, 110, 111, 1100, 1101, 11001, **11011**}
MooreMI	{λ, 0, 1, 10, 11, 011, 101, 110, 111, **11111**}

Fonte: Elaboração do autor (2020).

Na Tabela 10 apresentamos os critérios avaliados, são eles: o número de conjecturas (modelos) que foram construídos, o número de perguntas realizadas pelo algoritmo e o número de iterações realizadas ao longo da execução. Estes foram selecionados para avaliar o custo computacional. O comprimento da maior palavra executada (destacada na Tabela 11) é também um importante critério para a avaliação de desempenho de algoritmos de aprendizado de modelos, que, segundo (BALCÁZAR; DIAZ; GAVALDÁ, 1997), representa uma ineficiência do aprendizado. Todas as informações, que foram necessárias para cada um dos algoritmos convergirem corretamente para a solução do problema apresentado, encontram-se na Tabela 11.

Algumas ressalvas devem ser feitas. O algoritmo MooreMI implementa aprendizado *offline* (ver Seção 3.2), desta forma ele necessita de algum agente externo que forneça um conjunto completo de informações a respeito do sistema a ser modelado. Como descrito na Seção 3.2.4, o conjunto fornecido para a execução do exemplo foi insuficiente (ver Figura 33). Para a solução do problema, apresentamos a informação referente à palavra 11111. Mas isto foi possível apenas pela existência de um *professor* com o conhecimento sobre a linguagem, o que, no contexto da aplicação em questão, é desconhecido, inviabilizando sua aplicação direta na arquitetura LBT.

Os algoritmos L* e IDS, por implementarem procedimentos de aprendizado *online*, obtém iterativamente, informações acerca de novas cadeias de eventos. A cada iteração, facilitam uma análise qualitativa dos resultados apresentados nas Tabelas 10 e 11. Ambos os procedimentos encontraram uma solução no segundo modelo criado. A abordagem L* necessitou de uma iteração a menos que o IDS, bem como realizou menor número de perguntas ao professor. Fato este que resultou também na necessidade de um conjunto menor de informações por parte do L* (Tabela 11).

O que explica tal diferença é a forma com que as cadeias a serem solicitadas pelo aluno são definidas. O algoritmo IDS combina todos os elementos do conjunto de palavras T com todos os elementos do conjunto de sufixos V e, ao longo de sua execução, estes conjuntos vão aumentando, o que representa um aumento exponencial do número de cadeias estudadas ao longo de cada iteração do algoritmo (ver Seção 3.2.3).

Por outro lado, o algoritmo L* adiciona o evento causador da inconsistência ao conjunto E, resultando na adição de uma nova coluna à Tabela de Observação. No pior caso, esta nova coluna poderá exigir um número de perguntas igual ao número de linhas da Tabela de Observação. Este número de linhas da tabela aumenta em função da cardinalidade de S ($|S|$), lembrando que S é o número de linhas da parte superior da tabela, que na construção do ADEF, representa o número de estados deste autômato. Chegamos assim à Fórmula (1), que relaciona o número de linhas da Tabela de Observação (n) com as cardinalidades de S e do alfabeto, $|S|$ e $|\Sigma|$, respectivamente.

$$n = |S| + |S| \cdot |\Sigma| \qquad (1)$$

Desta forma, o número de cadeias a serem estudadas a cada inconsistência aumenta linearmente, enquanto que no algoritmo IDS ocorre aumento exponencial.

Isto conduziu-nos a utilizar um procedimento de aprendizado baseado no algoritmo L* no método LBT, proposto na próxima seção. Não podemos simplesmente utilizá-lo, pois frente a um sistema do qual não se tem o conhecimento de sua implementação, caso dos SIS, a realização da pergunta "é equivalente?" é impossível de ser respondida. No entanto, este algoritmo, quando inserido na arquitetura LBT, possui o *model checker* como principal gerador de perguntas ao SUT ("professor"). Perguntas estas que serão realizadas na forma de testes.

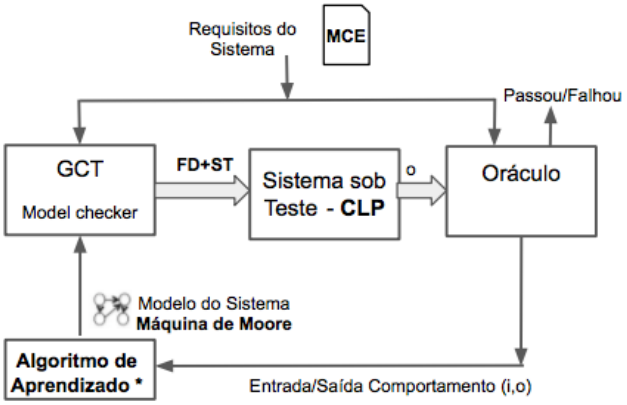
4 ADAPTAÇÃO DO MÉTODO LBT PARA SIS

Este capítulo apresenta uma proposta de método de Teste Baseado em Aprendizagem para verificar a conformidade de Sistemas Instrumentados de Segurança. Apresentamos as adaptações realizadas nos componentes da arquitetura *Learning-based Testing* (Seção 3) para possibilitar seu uso no domínio da aplicação. O método é ilustrado por um pequeno exemplo ao final do capítulo.

4.1 VISÃO GARAL DA ADAPTAÇÃO DO MÉTODO LBT

Em nossa proposta os Requisitos do Sistema são representados utilizando o formato Matriz de Causa e Efeito (MCE). A partir da MCE é realizada a geração automática de fórmulas lógicas. Um modelo inicial do sistema é obtido pelo Algoritmo de Aprendizado através das informações provenientes da execução do primeiro conjunto de testes. Os Requisitos são verificados no atual Modelo do Sistema pelo Gerador de Casos de Teste (GCT) Model Checker. Dois tipos de contraexemplos podem ser encontrados: Falha sob Demanda (FD - *Fail on Demand*) e Ativação Extemporânea (ST - *Spurious Trip*).

Figura 29 - LBT adaptado para SIS.



Fonte: Elaboração do autor (2020).

Os contraexemplos são executados como casos de teste no Sistema sob Teste. O comportamento do sistema frente aos testes é avaliado pelo Oráculo, responsável por identificar uma falha no comportamento do SUT (encerrando o procedimento) ou permitir a continuidade da execução do algoritmo. Neste último caso, o Algoritmo de Aprendizado recebe as

informações de Entrada/Saída do Comportamento do sistema da última rodada de testes para formular um novo Modelo do Sistema. Modelo este que será verificado pelo *Model Checker* para geração de novos casos de teste e assim por diante até o Oráculo identificar uma falha ou o algoritmo atingir o critério de parada.

4.2 CONCEITOS PRELIMINARES

Nesta seção, detalhamos como são descritas as especificações de segurança nas Matrizes de Causa e Efeito e as fórmulas de Lógica Temporal Linear.

4.2.1 Matrizes de Causa e Efeito

A norma IEC-62881 descreve um simples e vastamente aceito método para documentação de lógicas de intertravamento em indústrias de processo e manufatura - a Matriz de Causa e Efeito (MCE). As MCEs descrevem as relações entre as condições causais - causas - e as ações necessárias de saída - efeitos. O SIS de uma plataforma exploratória de petróleo pode ser composto por 100 a 150 MCEs, cada uma composta por 50 linhas e 50 colunas.

Figura 30 - Matriz Causa e Efeito.

			EFEITO	TAG	EA-1	FD-1	FC-1
				Equipamento	Alarme de Emergência	Chama Detectada	Chama Confirmada
CAUSA							
Equipamento	Votação	TAG					
Alarme Manual		A-1			X	A1	
Detector de Fumaça		SD-1			X	X	T10
Detector de Chama	1oo3	FFD-1				A1	
Detector de Chama	2oo3	FFD-2					X
Detector de Chama		FFD-3					

Fonte: Adaptado de (VEIGA, 2018).

As causas são representadas por sinais criados por sensores ou outras formas de informação; efeitos são ações automaticamente realizadas por atuadores (principalmente válvulas e motores) ou manualmente por operadores ou alarmes e mensagens providas aos

operadores. Ambas são relacionadas pelo conteúdo da matriz, conforme ilustrado na Figura 30 (IEC 62881, 2018).

A matriz é constituída pelas linhas (causas), colunas (efeitos) e suas células apresentam as relações entre as causas e os efeitos através de símbolos. O símbolo "X" representa o operador lógico "OR", o símbolo "A" o operador lógico "AND". A presença de temporizador é expressa através do símbolo "T" e votações também são expressas na tabela. Logo, a MCE apresentada deve ser interpretada da seguinte forma:

- EA-1 deve ser acionado quando A-1 ou SD-1 está ativo;
- FD-1 deve ser acionado quando SD-1 ou A-1 e ao menos uma das entradas do grupo FFD-1, FFD-2, FFD-3 está ativa;
- FC-1 deve ser acionado quando SD-1 permanece ativo por 10 segundo ou "2oo3" (*2 out of 3*) ao menos duas das entradas do grupo FFD-1, FFD-2, FFD-3 estão ativas.

A fim de permitir o uso das MCE já utilizadas na especificação dos SIS pelos engenheiros, adotamos a metodologia proposta em (DOS REIS, 2018) que traduz, sistematicamente, a MCE em fórmulas de lógica temporal linear (LTL).

4.2.2 Lógica Temporal Linear

A lógica temporal linear foi apresentada em (PNUELI, 1977) como um formalismo apropriado para a descrição de um histórico de execução de um programa sem a introdução explícita dos estados do programa ou do tempo. Além disso, importantes propriedades, como exclusão mútua, ausência de *deadlock* entre outras, podem ser elegantemente expressas (CLARKE; EMERSON; SISTLA, 1986).

As propriedades expressas em LTL são definidas por uma combinação de proposições atômicas, constantes, conectores booleanos e operadores temporais.

Sendo p e q proposições atômicas, os operadores temporais são:

- *Future* (F): F_P (denota que p será verdadeiro no futuro);
- *Generally* (G): G_P (denota que p sempre será verdadeiro);
- *Until* (U): $p\ U\ q$ (denota que p será verdade até que q seja verdade);
- *Next* (N): Np (denota que p será verdadeiro no próximo instante).

Os conectores booleanos são:

- *And* ($\wedge$): $p \wedge q$ (denota p e q);
- *Or* ($\vee$): $p \vee q$ (denota p ou q);
- *Not* ($\neg$):$\neg\, p$ (denota não p);
- Implicação ($\rightarrow$): $p \rightarrow q$ (denota se p então q);
- Bi-implicação ($\leftrightarrow$): $p \leftrightarrow q$ (denota p se e somente se q).

Com isso, para expressarmos que "sempre o motor estará ligado quando os sensores 1 e 2 estiverem sensibilizados", por exemplo, podemos considerar as seguintes proposições:

- Motor ligado (M);
- Sensor 1 sensibilizado ($S1$);
- Sensor 2 sensibilizado ($S2$).

A partir destas proposições, o exemplo pode ser expresso em LTL pela fórmula:

$$G((S1 \wedge S2) \rightarrow M) \qquad (2)$$

Estas fórmulas podem ser extraídas automaticamente das matrizes de causa e efeito, mantendo o engenheiro de projeto distante das formulas LTL, conforme explicado a seguir.

4.3 TRADUÇÃO DE MCE PARA FÓRMULAS LTL

Em nossa proposta de uso da arquitetura LBT no contexto dos SIS na IPG, utilizamos a representação em Matrizes de Causa e Efeito dos requisitos de segurança do SIS que são padronizadas na IEC-62881: 2018 e já utilizada na aplicação (ver Seção 2.2). Esta forma permite a análise modular de grupos de equipamentos, auxiliando, assim, na visualização de causas ou efeitos específicos. A partir da MCE suas informações são traduzidas para fórmulas de Lógica Temporal Linear (LTL).

Um conjunto de regras para tradução sistemática de MCEs em fórmulas LTL foram apresentadas por Dos Reis (2018), possibilitando que o método se adeque às linguagens e aos documentos já utilizados pelos engenheiros da IPG, bem como identifique dois tipos de falhas no sistema implementado: Falha sob Demanda (FD - *Fail on Demand*), caso em que há causa sem o efeito; e Ativação Extemporânea (ST - *Spurious Trip*), quando há o efeito sem a causa.

Para cada efeito da MCE, são definidas duas propriedades em LTL que especificam que a saída do sistema é livre de Falha sob Demanda (FDF - *fail on demand free*) e livre de

Ativação Extemporânea (STF - *spurious trip free*). Estas propriedades podem ser extraídas diretamente através das seguintes fórmulas:

$$FDF_{effect}: G \neg (cause \wedge \neg effect) \quad (3)$$

$$STF_{effect}: G \neg (\neg cause \wedge effect) \quad (4)$$

Onde:

$effect$ é a variável de saída a ser verificada, como, por exemplo, um sinal de ativação do alarme de incêndio; $cause$ é a variável que determina quando o efeito deve ser acionado, por exemplo, a ocorrência do incêndio.

Por exemplo, para a Matriz de Causa e Efeito apresentada na Figura 30, verificando o correto funcionamento do Alarme de Emergência $(EA-1)$, extraímos as fórmulas:

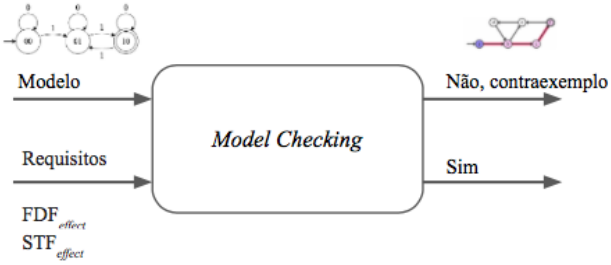
$$FDF_{EA-1}: G \neg ((A-1 \vee SD-1) \wedge \neg EA-1) \quad (5)$$

$$STF_{EA-1}: G \neg (\neg (A-1 \vee SD-1) \wedge EA-1) \quad (6)$$

4.4 MODEL CHECKING

O componente *model checking* receberá as fórmulas FDF_{effect} e STF_{effect}, juntamente com o modelo inferido pelo Algoritmo de Aprendizado para então, verificar se o modelo satisfaz as propriedades. Caso o modelo não satisfaça alguma propriedade, o verificador apresenta um contraexemplo. Caso contrário, o *model checker* informará que os requisitos são cumpridos pelo modelo, conforme ilustrado na Figura 31.

Figura 31 – Entradas e saídas do verificador.



Fonte: Elaboração do autor (2020).

O procedimento de *model checking* na arquitetura LBT é o principal provedor dos casos de teste aplicados ao SUT. Seus contraexemplos vão refinando o modelo através do algoritmo de aprendizado e identificando as falhas do sistema por meio do Oráculo (ver Seção 3). Detalhamos a seguir o algoritmo de aprendizado elencado para a presente proposta.

4.5 ALGORITMO DE APRENDIZADO

O algoritmo de aprendizado de modelos que adaptamos para a utilização na arquitetura LBT foi o algoritmo L*, fundamentado na análise comparativa das abordagens estudadas (Seção 3.2.5). Ainda assim, realizamos uma série de alterações, as quais detalhamos nesta seção.

A mudança mais significativa fica por conta da linguagem de modelagem utilizada. O modelo aprendido será expresso na forma de um autômato de Moore. A saída de cada estado neste autômato de Moore contém os valores individuais dos sensores e atuadores, para tanto, propomos a utilização de um algoritmo auxiliar para realizar a leitura dos níveis lógicos das variáveis disponíveis do SUT (CLP). Os casos de teste são representados como sequências de eventos no autômato.

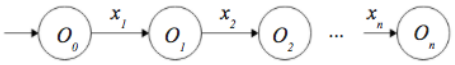
Conforme podemos verificar na Seção 3.2.5, a cardinalidade do alfabeto tem forte influência no aumento do conjunto de teste. Desta maneira, uma abstração dos possíveis casos de teste pode ser utilizada para reduzir o alfabeto do modelo. Por exemplo, para um sistema com dois sensores podemos modelar o comportamento deles através de eventos de sensibilização e dessensibilização de cada um, totalizando quatro símbolos. Uma abstração pode utilizar apenas um evento para simbolizar a alternância do estado de cada um dos sensores, utilizando dois símbolos.

As perguntas na forma "x pertence ao conjunto?", sendo x uma cadeia de eventos, realizadas no procedimento original, foram analogamente utilizadas na forma de execução do teste x no CLP. Já as perguntas de equivalência são impraticáveis, tendo em vista que não temos acesso à implementação do SUT. Uma alternativa pode ser a substituição do teste de equivalência por um conjunto de testes de pertinência (embora não seja uma forma equivalente).

O procedimento de inferência do modelo demanda que o usuário forneça um conjunto inicial de palavras, de qualquer tamanho, a serem testadas (i_0), e defina as variáveis do sistema que serão atreladas ao alfabeto de saída (O) do autômato. Os testes são executados um a um, evento a evento ($x_1 x_2 \ldots x_n$), e a sequência das variáveis de saída do autômato, obtidas da

observação do SUT, pelo algoritmo auxiliar, definirão os estados do modelo ($O_0O_1O_2 \dots O_n$). O algoritmo traduz estas informações para o modelo na forma apresentada na Figura 32.

Figura 32 - Modelo $i\_0 = \{x_1x_2 \dots x_n\}$ Saídas: $\{O_0O_1 \dots O_n\}$.



Fonte: Elaboração do autor (2020).

Utilizamos o procedimento *Complete* para completar o modelo a fim de excluir a necessidade de modelar um conjunto completo de pares entrada e saída, conforme definido na Seção 3.2.2.

Isto é encontrado no algoritmo MooreMI (Seção 3.2.4). No entanto, a abordagem de completar com autolaços a Máquina de Moore que é utilizada em MooreMI não contribui para o processo de aprendizado do modelo. Portanto, propomos uma nova maneira de completar o autômato, na qual são criados estados para expressar a indeterminação das variáveis do Sistema sob Teste.

O procedimento *Complete*, identifica todos os eventos de todos os estados do autômato que ainda não estão definidos. Para estes, é definido um estado de destino verificando no alfabeto de saída (O), quais variáveis da entrada do CLP pertencem a O. Para estas, com base na abstração utilizada para definir os eventos do modelo, seu estado é conhecido e é utilizado na representação do estado. Já para as variáveis que não se tem conhecimento sobre seu estado lógico, são representados pelo símbolo *. Por exemplo: para $O = (a, b, c)$, sabendo-se que b encontrar-se-á em valor lógico "0" e desconhecendo o estado das variáveis a e c, o estado do modelo será - *0*. O Algoritmo 6 apresenta o pseudocódigo do algoritmo de aprendizado de autômatos e do procedimento *Complete*.

Algoritmo 6 - Pseudoalgoritmo de aprendizado de autômato.

```
1.Define: Aprendizado(SUT: sistema sob teste, i_0: conjunto de teste inicial, I:
alfabeto de entrada, O: alfabeto de saída)
2. t_0 = nova pilha (i_0)
3. Estado_atual = Ler_SUT (O)
4. q0 = Estado_atual
5. Cria_estado(q0)
6. Enquanto t_0 não vazia
7.t = pop(t_0)
8.Reset(SUT)
```

```
9.Estado_atual = q0
10.Para todo a ∈ I e a ∈ t:
11.Testa_SUT (a)
12.Estado_anterior = Estado_atual
13.Estado_atual = Ler_SUT (O)
14.Se não existe (Estado_atual, modelo)
15.Cria_estado(Estado_atual, modelo)
16.Cria_evento(δ(Estado_anterior, a) = Estado_atual, modelo)
17. Enquanto não_deterministico(modelo)
18.Estado_causa = não_determinístico(modelo)
19.modelo = split(Estado_causa, modelo)
20.refina(modelo)
21. Se é_completo (modelo)
22.Retorna modelo
23. Senão
24.Complete (modelo)
25. Fim
26. Define: Complete(modelo)
27. Para todo q ∈ modelo.Q
28.Para todo b ∈ modelo.Σ
29.Se indefinido(δ(q, b))
30.Estado_anterior = q
31.Estado_atual = λ
32.Para todo a ∈ alfabeto_de_saída
33.Se estado da variável a é deduzido a partir de δ(q, b)
34.Estado_atual = Estado_atual. a
35.Senão
36.Estado_atual = Estado_atual. *
37.Se não existe(Estado_atual,modelo)
38.Cria_estado(Estado_atual)
39.δ(q, b) = Estado_atual
40. Retorna modelo
41. Fim
```

Fonte: Elaboração do autor (2020).

A rotina de aprendizado do modelo inicia recebendo o SUT, um conjunto de teste e os alfabetos de entrada e saída (Linha 1). Na linha 2 é definida uma pilha (t_0) com o conjunto de teste inicial (i_0). Na linha 3 o algoritmo auxiliar faz a leitura das variáveis do SUT relacionadas ao alfabeto de saída do modelo (O) e, a partir desta modela, na linha 4, o estado inicial. Das linhas 6 a 16 o conjunto de teste é executado. A cada teste concluído, o SUT é reiniciado (linha 8). O teste vai sendo executado evento a evento (linha 11), e o estado do SUT é observado e

retratado pelo estado atual do modelo (linha 13). Caso o atual estado não faz parte ainda do modelo (linha 14), este será criado na linha 15. A transição é modelada na linha 16.

Na linha 18 é verificado se há não determinismo no modelo construído, ou seja, a partir de um estado atual, observamos estados futuros diferentes. Isto significando que o estado atual representa, na verdade, mais de um estado do SUT e, por este motivo, deve ser dividido (linha 19). Na Seção 4.6, dois exemplos são executados detalhadamente. Um não determinismo apresentado na Figura 53 surge e elucida o procedimento de refinamento do modelo (linha 20).

Com o modelo determinístico, é verificado na linha 21 se o modelo obtido é completo. Caso afirmativo, o algoritmo conclui sua execução apresentando o modelo (linha 22). Ao passo que quando incompleto, o modelo será completado (linha 25).

O procedimento de completar o modelo identifica todos os eventos de todos os estados do autômato que ainda não estão definidos (linhas 27 a 29). Para estes, é definido o estado de destino verificando no alfabeto de saída (O) quais variáveis têm seu estado lógico conhecido sem a execução do teste no SUT. Estes são utilizados na representação do estado (linha 34). Já para as variáveis que não se tem conhecimento sobre seu estado lógico, são representados pelo símbolo * (linha 36). Uma vez definidos todos os eventos do modelo, ele está completo e será apresentado (linha 40), encerrando assim a execução do algoritmo.

Apresentamos a seguir a aplicação do método LBT proposto neste capítulo, para um exemplo ilustrativo.

4.6 EXEMPLO ILUSTRATIVO

Nesta seção, apresentamos um exemplo didático para ilustrar o método proposto. Este exemplo consiste de um sistema implementado que não cumpre com a especificação de segurança em um caso especifico em que um tradicional método de geração de casos de teste não é capaz de identificar a imperfeição. A inconformidade do sistema foi atrelada a um elemento de memória, responsável por definir o estado atual do sistema não apenas com base nos sinais de entrada, mas considerando também seu estado anterior. Este elemento de memória é característico de Sistemas Instrumentados de Segurança. O sistema simplificado é constituído por dois sensores e um atuador. A especificação de segurança, fornecida na forma de MCE (Figura 33), explicita que o Atuador ($A1$) deve ser acionado sempre que o Sensor 1 ($S1$) ou o Sensor 2 ($S2$) estiverem sensibilizados.

Figura 33 - Matriz Causa e Efeito Especificação.

		EFEITO	TAG	A1
			Equipamento	Atuador
CAUSA				
Equipamento	TAG			
Sensor 1	S1			X
Sensor 2	S2			X

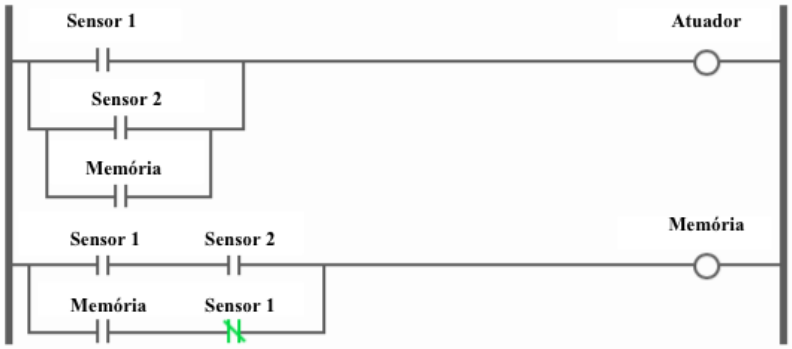
Fonte: Elaboração do autor (2020).

A partir da MCE extraímos algoritmicamente, conforme apresentado na Seção 4.3, as especificações expressas pelas fórmulas em LTL:

$$FDF_{A1}: G \neg ((S1 \vee S2) \wedge \neg A1) \quad (7)$$

$$STF_{A1}: G \neg (\neg (S1 \vee S2) \wedge A1) \quad (8)$$

Figura 34 - Diagrama *Ladder* da Implementação.



Fonte: Elaboração do autor (2020).

No entanto, há um erro na implementação do SUT em relação à especificação: o estado da memória ($M1$) afeta o estado de $A1$. Conforme apresentado no Diagrama *Ladder* da Implementação (Figura 34), uma vez que ambos os sensores estejam sensibilizados concomitantemente, $A1$ não será mais desacionado até que no mesmo instante $S1$ esteja ativo e S2 não. Essa variável interna de memória ($M1$) consideramos que é acessível, como de fato ocorre no contexto de aplicação. Algumas variáveis internas são monitoradas com o intuito de permitir a detecção precoce de eventuais falhas, quando ainda estão em *software*.

Para a modelagem do sistema, consideramos reduzir o alfabeto, conforme abordado e aplicado em (PROVOST; ROUSSEL; FAURE, 2014). Sendo assim, definimos dois eventos externos, o evento F_1 ("*Flip Sensor 1*"), que inverte estado de $S1$, e o evento F_2 ("*Flip Sensor 2*"), o mesmo para $S2$. Uma vez os sensores, inicialmente, dessensibilizados a ocorrência do

evento F_1, representa a sensibilização de $S1$. O evento F_2 provoca a sensibilização de $S2$ e um novo evento F_2 dessensibiliza ele.

Apresentamos as abordagens aplicadas ao problema proposto na Tabela 12. São detalhadas as variáveis utilizadas como saída para o autômato de Moore, juntamente com os conjuntos iniciais de teste adotados nas Propostas 1 e 2. Os valores lógicos das variáveis de saída representam os estados do autômato.

Tabela 12 - Propostas Executadas.

Proposta	Representação	Conjunto Inicial de Teste
Proposta 1	(S1, S2, A1, M1)	$\{F_1, F_2\}$
Proposta 2	(S1, S2, A1)	$\{F_1, F_2\}$

Fonte: Elaboração do autor (2020).

Por exemplo, na Proposta 1 o estado 0101 denota que, para este estado encontram-se, respectivamente, $S1$ dessensibilizado, $S2$ sensibilizado, $A1$ inativo e a variável $M1$ com valor lógico "1". Na Proposta 2, o estado é representado apenas por três variáveis, as mesmas da Proposta 1 excluindo a variável $M1$. Desta forma o estado 101 do autômato representa, respectivamente, $S1$ sensibilizado, S2 dessensibilizado e $A1$ ativo, não distinguindo, então, nesta representação de estado o valor da variável $M1$.

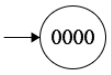
Na coluna "Conjunto Inicial de Teste" da tabela apresentamos os casos de teste executados para a geração do primeiro modelo do SUT, os quais são dois. O primeiro teste será a execução do evento F_1, que, conforme apresentado anteriormente, inverte o estado de $S1$. Já o segundo teste é a execução do evento F_2, evento análogo para $S2$. Na representação $\{F_1, F_2\}$ apresentamos um conjunto com dois casos de teste. Ao passo que $\{F_1F_2\}$ será utilizado para representar um único caso de teste que é composto pelo evento F_1 concatenado com o evento F_2.

Estas variações foram executadas individualmente e os modelos construídos a cada iteração são apresentados sequencialmente a seguir.

4.6.1 Execução Proposta 1

O procedimento inicia observando o estado inicial do SUT e através do algoritmo auxiliar aprendendo o modelo (Figura 35).

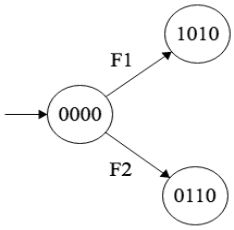
Figura 35 - Estado Inicial Observado.



Fonte: Elaboração do autor (2020).

A execução do conjunto de teste inicial $\{F_1, F_2\}$ é então realizada por parte do algoritmo de aprendizado, que constrói o modelo apresentado na Figura 36.

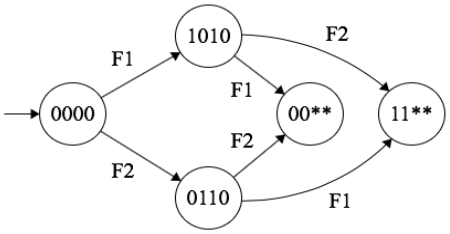
Figura 36 - P1 - Modelo 1a - $\{F_1, F_2\}$.



Fonte: Elaboração do autor (2020).

No entanto, os estados 1010 e 0110 não possuem os arcos definidos, portanto o algoritmo de aprendizado irá completar a máquina de Moore para a ocorrência dos eventos F_1 e F_2 com as informações que possui, sem execução de novos testes no SUT. O algoritmo tem o conhecimento que F_1 e F_2 alteram o estado lógico das entradas $S1$ e $S2$. Estas variáveis pertencem ao conjunto O de saída do modelo, no entanto, pertencem à O as variáveis $A1$ e $M1$ também. Porém o estado destas variáveis só pode ser definido executando-se a cadeia de eventos no SUT. Ao invés de executar os testes, o algoritmo define os estados utilizando o símbolo * para expressar a indeterminação (ver Seção 4.5). Construindo, desta forma, o modelo apresentado a seguir (Figura 37).

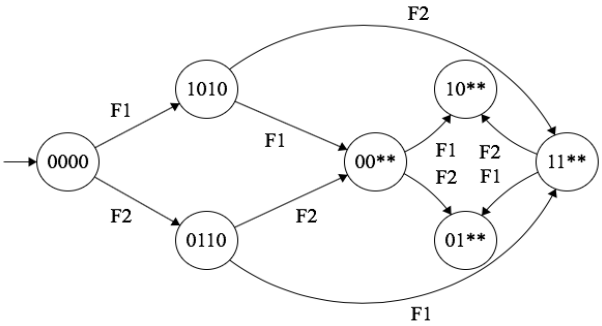
Figura 37 - P1 - Modelo 1b - *Completa* (1a).



Fonte: Elaboração do autor (2020).

Os estados 00** e 11** surgiram ao serem definidas as transições a partir de 1010 e 0110, no entanto o modelo ainda é incompleto, uma vez que não estão definidas as transições a partir dos novos estados. O algoritmo atua, então, na definição das transições a partir de 00** e 11** para completar o modelo atual, conforme apresentado na Figura 38.

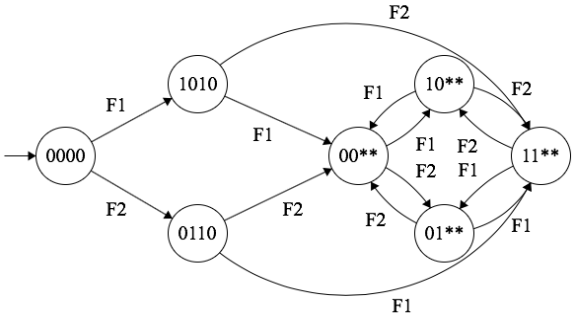
Figura 38 - P1 - Modelo 1c - *Completa* (1b).



Fonte: Elaboração do autor (2020).

Foram criados mais dois estados, 01** e 10**. Estados estes que fazem do novo modelo ainda incompleto, o procedimento para completar o modelo é executado novamente, resultando, agora, no modelo inicial completo apresentado na Figura 39.

Figura 39 - P1 - Modelo 1 - *Completa* (1c).



Fonte: Elaboração do autor (2020).

Com o Modelo 1 completo, o procedimento de *model checking* recebe a especificação expressa em LTL que foi extraída diretamente da matriz de causa e efeito e o modelo construído pelo algoritmo de aprendizado. O *model checker* busca no espaço de estados do modelo por contraexemplos para as duas especificações (Fórmulas 9 e 10), que representam, respectivamente, a FD e a ST:

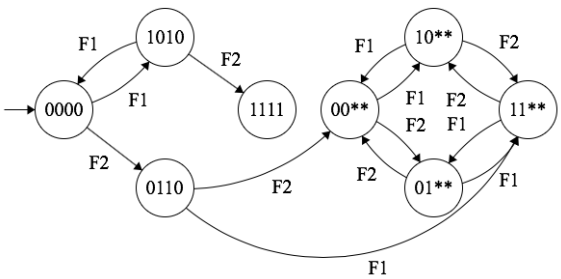
$$G \neg ((S1 \vee S2) \wedge \neg A1) \quad (9)$$

$$G \neg (\neg(S1 \vee S2) \wedge A1) \quad (10)$$

O algoritmo retorna como contraexemplos para FD e ST, respectivamente, F_1F_2 e F_1F_1. A primeira cadeia leva o modelo ao estado 11**, representando que $S1$ e $S2$ estão sensibilizados, ou seja, a demanda pelo atuador ocorreu. No entanto, a não determinação do estado de A representa que ele poderá ou não atuar configurando uma FD. Já F_1F_1 conduz o modelo ao estado 00**, representando que $S1$ e $S2$ estão dessensibilizados, ou seja, a demanda não ocorreu. A indeterminação do estado do atuador possibilita que ocorra uma ST.

Pelo fato de os contraexemplos serem originados de estados com indeterminações, o modelo deve ser refinado, pois é um erro de modelagem. Estes contraexemplos providos pelo *model checker* são os novos casos de teste que serão aplicados ao SUT. Executando a cadeia F_1F_2, são observadas as variáveis $S1$, $S2$, $A1$ e $M1$ e seus estados lógicos criam o novo estado 1111. Da cadeia F_1F_1 obtém-se $O = 0000$, fazendo com que o novo modelo (Figura 40) seja obtido.

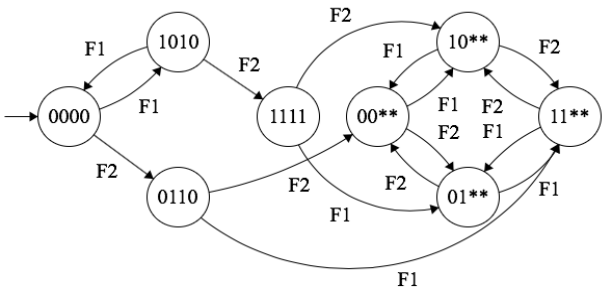
Figura 40 - P1 - Modelo 2a - $\{F_1F_2, F_1F_1\}$.



Fonte: Elaboração do autor (2020).

As transições partindo do estado 1111, devem ser definidas para os eventos F_1 e F_2 para completar o Modelo 2a. Completando como descrito na iteração anterior, o Modelo 2 é obtido.

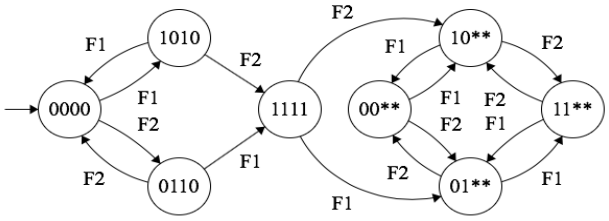
Figura 41 - P1 - Modelo 2 - *Completa* (2a).



Fonte: Elaboração do autor (2020).

O Modelo 2 é uma máquina de Moore completa, que, agora, será verificada pelo *model checker* a fim de identificar possíveis FD e ST. O procedimento retornou como contraexemplo as cadeias F_2F_2 e F_2F_1 que levam, respectivamente, a estados que representam FD (estado 11**) e ST (estado 00**). Estas cadeias de eventos são então executadas no SUT como um novo conjunto de teste. As variáveis observadas no SUT representam o estado 0000 para o teste F_2F_2 e 1111 para F_2F_1, fazendo com que seja construído o Modelo 3.

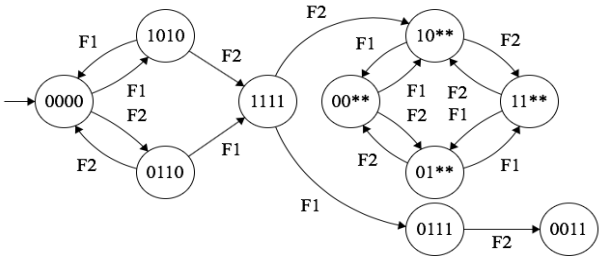
Figura 42 - P1 - Modelo 3 - $\{F_2F_2, F_2F_1\}$.



Fonte: Elaboração do autor (2020).

O Modelo 3 é completo, desta forma o *model checker* efetua a busca por FD e ST. O procedimento apresentou as cadeias $F_1F_2F_1$ e $F_1F_2F_1F_2$ como contraexemplos, estes alcançando respectivamente os estados 01** (FD) e 00** (ST). Os contraexemplos são então executados como teste no SUT, e as saídas obtidas em cada uma delas são utilizadas no refinamento do modelo, apresentado na Figura 43.

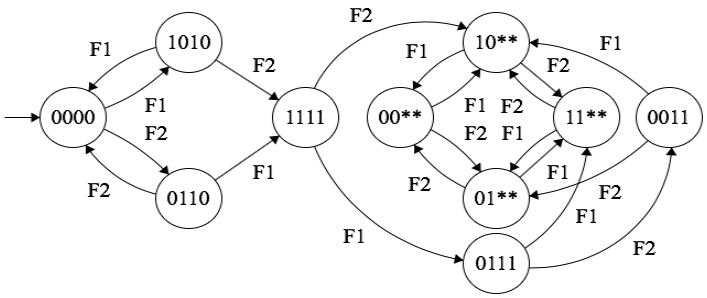
Figura 43 - P1 - Modelo 4a - $\{F_1F_2F_1, F_1F_2F_1F_2\}$.



Fonte: Elaboração do autor (2020).

No entanto, o presente modelo não é completo. O procedimento de completá-lo é, então, executado, resultando no Modelo 4, apresentado na Figura 44.

Figura 44 - P1 - Modelo 4 - *Completa* (4a).



Fonte: Elaboração do autor (2020).

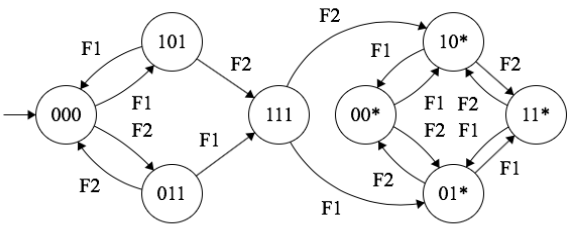
Com o Modelo 4 – completo – o *model checking* busca por estados que descumpram as propriedades de segurança. E encontra o estado 0011, que denota que os sensores $S1$ e $S2$ estão dessensibilizados e o atuador e o elemento de memória estão ativos. Como a cadeia de eventos $F_1F_2F_1F_2$ foi um contraexemplo gerado pelo *model checking* na iteração anterior e já testado no SUT. Desta forma, o oráculo confirma que é uma falha de implementação e fornece este veredicto, encerrando a execução do algoritmo. Este foi um caso de ativação extemporânea onde não temos a causa, porém o sistema de segurança atua mesmo assim.

4.6.2 Execução Proposta 2

Executamos agora a abordagem LBT adaptada para o mesmo exemplo ilustrativo descrito anteriormente nesta seção, porém alterando as variáveis atreladas à saída da máquina de Moore. No que chamamos de Proposta 2 (ver Tabela 12) na qual excluímos a variável de Memória do alfabeto de saída do modelo.

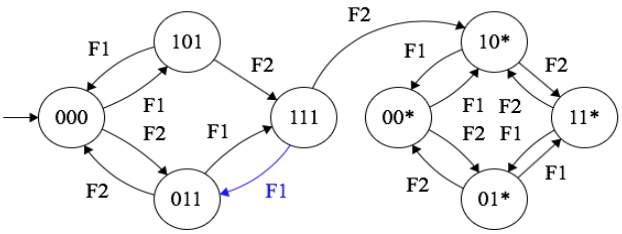
Nesta Proposta a execução decorreu de forma semelhante à Proposta 1, gerando uma sequência de modelos com estrutura idêntica aos modelos gerados na proposta anterior. Distinguindo-se apenas pela ausência do quarto dígito do alfabeto de saída do modelo. Quando o algoritmo constrói o terceiro modelo completo (Figura 45), análogo ao Modelo 3 encontrado na Proposta 1 (Figura 42), o procedimento de *model checking* encontra as cadeias $F_1F_2F_1$ e $F_1F_2F_1F_2$ como contraexemplos, exatamente como ocorreu na execução da proposta anterior.

Figura 45 - P2 - Modelo 3 (análogo Figura 34).



Fonte: Elaboração do autor (2020).

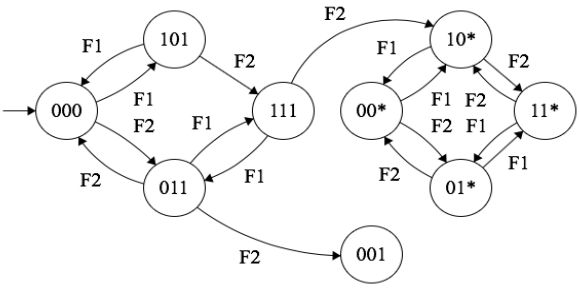
Figura 46 - Modelo 4a $\{F_1F_2F_1\}$.



Fonte: Elaboração do autor (2020).

Quando o algoritmo modela a transição destacada na Figura 46 (ao executar o contraexemplo $F_1F_2F_1$), ele faz com que a execução de $F_1F_2F_1F_2$ resulte em um não-determinismo no Modelo 4b, apresentado na Figura 47.

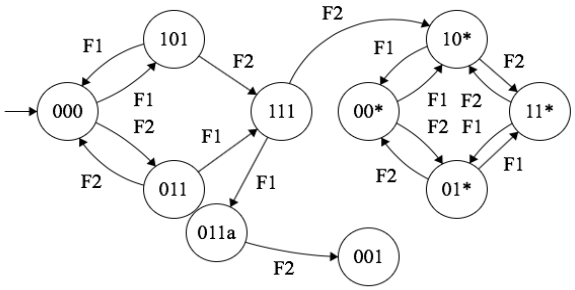
Figura 47 - Modelo 4b $\{F_1F_2F_1F_2\}$.



Fonte: Elaboração do autor (2020).

O procedimento de *split* é então executado no estado (011) causador do indeterminismo, originando o estado $011a$. O modelo obtido é apresentado na Figura 48.

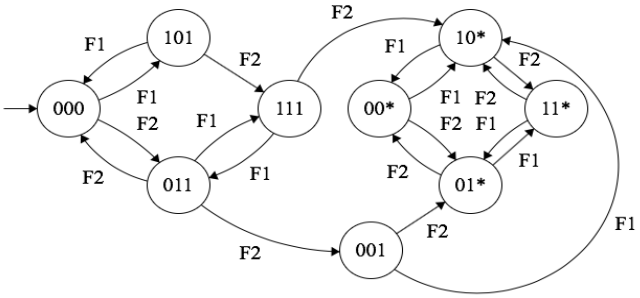
Figura 48 - Modelo 4c - *Split*(011).



Fonte: Elaboração do autor (2020).

O Modelo 4c é determinístico, no entanto, incompleto. Logo, o procedimento de completar é realizado construindo o Modelo 4 (Figura 49).

Figura 49 - Modelo 4 - *Complete*(4c).



Fonte: Elaboração do autor (2020).

O teste de sequência $F_1F_2F_1F_2$ encontra a falha de implementação com o sistema alcançando um estado onde ambos os sensores estão dessensibilizados e o atuador encontra-se acionado, modelado pelo estado 001. Desta forma, o algoritmo encerra sua execução, bem como aconteceu na Proposta 1 (Seção 4.6.1).

Do resultado da execução das 2 propostas foi possível identificar que para o caso aplicado, a definição do alfabeto de saída da máquina de Moore não influenciou no resultado final do procedimento de verificação do sistema. Isto foi possibilitado pelo procedimento de Split, que frente ao não determinismo encontrado no Modelo 4b criou o estado $001a$. Estado

este que identificou o elemento de memória que foi explicitado no alfabeto de saída da Proposta 1 e não no alfabeto da Proposta 2.

A fim de possibilitar uma avaliação do método proposto frente ao contexto atual dos SIS na IPG, aplicamos a seguir, a estratégia CEG-BOR ao problema ilustrativo apresentado na Seção 4.6.

4.6.3 Execução CEG-BOR

O algoritmo *Cause-Effect Graphing - Boolean Operator* (CEG-BOR) traduz especificações representadas em lógica proposicional para Grafos Causa e Efeito (CEG). Através destes CEGs o procedimento visita todos os nodos e por meio de regras (Algoritmo 7) definindo os casos a serem testados. É uma abordagem projetada para encontrar falhas em operadores Booleanos (BOR).

Algoritmo 7 – Regras algoritmo CEG-BOR.

Assumindo que $CEGBOR(N_1)$ e $CEGBOR(N_2)$ são conjuntos de teste constantes para os nodos N_1 e N_2 respectivamente. $CEGBOR(N)$ será o conjunto de teste constante gerado para N. Então,

Regra 1: Se N é um nodo causa, $CEGBOR(N)$ é dado por $\{(t),(f)\}$.

Regra 2: Se N = $N_1 \wedge N_2$, $CEGBOR(N)$ é construído como segue:

$CEGBOR_t(N) = CEGBOR_t(N_1) \otimes CEGBOR_t(N_2)$,

$CEGBOR_f(N) = (CEGBOR_f(N_1) \times \{t_{N2}\}) \cup (\{t_{N1}\} \times CEGBOR_f(N_2))$,

onde $t_{N1} \in CEGBOR_t(N_1)$, $t_{N2} \in CEGBOR_t(N_2)$, e $(t_{N1}, t_{N2}) \in CEGBOR_t(N)$.

Regra 3: Se N = $N_1 \vee N_2$, $CEGBOR(N)$ é construído como segue:

$CEGBOR_f(N) = CEGBOR_f(N_1) \otimes CEGBOR_f(N_2)$,

$CEGBOR_t(N) = (CEGBOR_t(N_1) \times \{f_{N2}\}) \cup (\{f_{N1}\} \times CEGBOR_t(N_2))$,

onde $f_{N1} \in CEGBOR_f(N_1)$, $f_{N2} \in CEGBOR_f(N_2)$, e $(f_{N1}, f_{N2}) \in CEGBOR_f(N)$.

Regra 4: Se N = $\neg N_1$, $CEGBOR(N)$ é construído como segue:

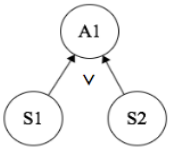
$CEGBOR_f(N) = CEGBOR_t(N_1)$,

$CEGBOR_t(N) = CEGBOR_f(N_1)$.

Fonte: Adaptado de (PARADKAR; 1997).

Importante salientar que estamos utilizando os mesmos SUT e MCE executados nas Propostas 1 e 2 para a aplicação do algoritmo CEG-BOR. Inicialmente, o procedimento irá receber as Fórmulas 6 e 7, que representam as especificações em LTL, extraídas da MCE (Figura 25). A partir delas, o grafo causa efeito é construído (Figura 50).

Figura 50 – CEG ($A1 = S1 \vee S2$).



Fonte: Elaboração do autor (2020).

O algoritmo inicia visitando os nodos $S1$ e $S2$ definindo de acordo com a regra 1 o conjunto $\{(t),(f)\}$ para cada um. Então, $A1$ é visitado, através da regra 3 são definidos os conjuntos $CEGBOR_t(A1) = \{(t,f),(f,t)\}$ e $CEGBOR_t(A1) = \{(f,f)\}$. Desta forma, estes são os casos de teste gerados pela estratégia. Apresentamos na Tabela 13.

Tabela 13 - Conjunto de teste - CEG-BOR.

S1, S2	A1
f, f	*F*
t, f	*T*
f, t	*T*

Fonte: Elaboração do autor (2020).

Os três casos de teste gerados pelo algoritmo CEG-BOR não são suficientes para encontrar a inconformidade do SUT.

4.6.4 Análise dos Resultados

No presente capítulo foram executadas duas variações do método proposto e a comparação com o método CEG-BOR para um exemplo ilustrativo. Tal exemplo apresentando um erro de implementação associado a um elemento de memória.

As Propostas 1 e 2 identificaram a inconformidade existente no SUT, ambas executando os mesmos casos de teste. Foram oito casos testados, a maior execução composta por quatro eventos. Já algoritmo CEG-BOR gerou apenas três casos de teste, uma cadeia de eventos vazia e uma execução de comprimento um com cada evento do sistema.

Para Sistemas Instrumentados de Segurança, a grande importância está na identificação de qualquer inconformidade. No entanto uma limitação do método de Teste Baseado em Aprendizagem proposto encontra-se na criação de um estado para cada possível combinação de valores lógicos de entradas do sistema. E isto se dá desde sua primeira iteração, caso a dimensão dos SIS conduzir a uma explosão combinatória, nenhum teste será executado e nenhuma informação será obtida do sistema implementado.

A partir do resultado da abordagem CEG-BOR frente a um exemplo de considerável simplicidade, observamos uma limitação desta técnica. O elemento de memória presente no SUT não é identificado pela técnica, e isto se dá pelo fato de os casos de teste serem provenientes apenas dos requisitos do sistema.

Foi o laço existente na arquitetura de Teste Baseado em Aprendizagem que é utilizado para refinar o modelo do sistema e, principalmente, gerar iterativamente novos casos de teste, que possibilitou às Propostas 1 e 2 a identificarem a inconformidade existente na implementação do SUT. Este arranjo pode ser uma alternativa para o uso da técnica CEG-BOR com mais eficiência.

5 CONCLUSÕES E PERSPECTIVAS

Acidentes em plataformas exploratórias de petróleo podem levar a danos irreversíveis à humanidade. Devido ao impacto potencial desses acidentes, os Sistemas Instrumentados de Segurança devem ter seu funcionamento validado com rigor.

O procedimento de validação por meio de testes de conformidade utilizado pela Petrobras não faz uso de métodos formais, conforme sugere a norma IEC-61511, a qual rege as boas práticas em SIS para a indústria de processos.

Por outro lado, a abordagem de *model checking* – formal – necessita de uma correta modelagem do sistema para, através dela, validar efetivamente o SIS. Modelagem essa ainda suscetível à explosão combinatória, devido à busca exaustiva no espaço de estados, uma característica dos algoritmos de *model checking*.

Com a finalidade de prospectar uma alternativa que contorne as limitações do uso de testes de conformidade e dos algoritmos de *model checking* que, principalmente, aumente a confiabilidade do procedimento de validação dos Sistemas Instrumentados de Segurança, estudamos a arquitetura de Teste baseado em Aprendizagem proposta por Meinke.

O Teste Baseado em Aprendizagem substitui a dificuldade de encontrar diretamente um modelo pela dificuldade de encontrar um conjunto de estradas/saídas que consigam representar o comportamento da planta. Para a aplicação do método a SIS, propusemos, a partir da sua metodologia de desenvolvimento, alterações específicas para aumentar a confiabilidade do procedimento de validação destes sistemas.

Propusemos a utilização direta das Matrizes de Causa e Efeito - documento regido pela norma IEC-62881, ao invés de fórmulas em LTL, a segregação dos contraexemplos do *model checking* em Falha sob Demanda e Ativação Extemporânea e a modelagem por meio de Máquina de Moore.

A adaptação do método de Teste Baseado em Aprendizagem realizada neste trabalho foi comparada ao algoritmo CEG-BOR, frente a um exemplo ilustrativo específico em que este não consegue encontrar o erro existente na implementação. O método proposto identificou a inconformidade, no entanto, a importante limitação do uso de LBT na validação de sistemas continua sendo a falta de garantia da correta implementação, já presente no método de validação atualmente utilizado.

A partir dos resultados obtidos, notamos que a arquitetura de Teste Baseado em Aprendizagem conseguiu, por meio da combinação dos algoritmos de *model checking* e de aprendizado de modelos, gerar e executar testes de conformidade que possibilitaram a identificação de uma falha presente no sistema de pequena dimensão aplicado.

O componente responsável pelo desempenho do método é o algoritmo de aprendizado de modelos. Por este motivo, ele deve ser explorado e aprimorado. É através dele que o algoritmo de *model checking* consegue gerar casos de teste mais eficazes. A apresentação didática de três algoritmos com exemplos executados passo a passo facilita seu entendimento, bem como apresenta alternativas distintas tanto no que diz respeito às linguagens de modelagem, quanto às maneiras de construir e refinar os modelos.

O procedimento de completar o modelo, utilizando estados para a representação de indeterminações proposto neste trabalho, ao invés da realização de novos testes, é uma abordagem que deve ser aperfeiçoada. O aproveitamento da série de modelos matemáticos que são gerados pela técnica deve ser estudada.

REFERÊNCIAS

ANGLUIN, D. Learning Regular Sets from Queries and Counterexamples*. **Information and Computation 75**, p. 87-106, 1987.

ANP. **Produção brasileira de petróleo em março cresce 2,8% em relação a fevereiro.** Publicado: 03 de maio de 2019. Disponível em: http://www.anp.gov.br/noticias/anp-e-p/5162-producao-brasileira-de-petroleo-em-marco-cresce-2-8-em-relacao-a-fevereiro. Acessado em: 02/06/2019 ás 12:03. Agência Nacional do Petróleo, Gás Natural e Biocombustíveis, 2019.

BALCÁZAR, J. L.; DIAZ, J.; GAVALDÁ, R. Algorithms for Learning Finite Automata from Queries: A Unified View. **Advances in Algorithms, Languages and Complexity**. Kluwer Academic Publichers, p. 53-72, 1997.

BENNACEUR, A.; MEINKE, K. Machine Learning for Software Analysis: Models, Methods, and Applications, **LNCS 11026**, p. 3-49, 2018.

CLARKE, E.M.; EMERSON, E.A.; SISTLA,A.P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, **ACM Transactions on Programming Languages and Systems**, vol. 8, no.2, p. 244-263, 1986.

CLARKE, E. M. et al. Model Checking: Algorithmic Verification and Debugging. **Turing Lecture.** Communications of the acm, p. 75-84, november 2009.

CARMEL, D; MARKOVITCH, S. Learning Models of Intelligent Agents. **AAAI-96 Proceedings**, 1996.

DOS REIS, L. P. E. et al. Verificação Formal de Sistemas Instrumentados de Segurança na Indústria de Petróleo e Gás Natural. **CBA 2018.**

EIA. Independent Statistics & Analysis. **Offshore production nearly 30% of global crude oil output in 2015.** Disponível em: https://www.eia.gov/todayinenergy/detail.php?id=28492. Acessado em: 02/06/2019 ás 12:11. U.S. Energy Information Administration, 2016.

FREY, G; LITZ, L. Formal methods in PLC programming. **International Conference on Systems, Man and Cybernetics**, 2000.

GERGELY, E. I.; COROIU, L.; POPENTIU-VLADICESCU, F. Methods for Validation of PLC Systems. **Journal Of Computer Science And Control Systems.** Oradea, p. 47-52. Maio 2011.

GIANTAMIDIS, G.; TRIPAKIS, S. Learning Moore Machines from Input-Output Traces. FM 2016, p. 291-309, 2016.

GRUHN, P.; CHEDDIE, H. L. Safety instrumented systems - design, analysis, and justifications. **ISA**, 2006.

IEC 61511. INTERNATIONAL STANDARD: Functional safety – Safety instrumented systems for the process industry sector. 2018.

IEC 62881. INTERNACIONAL STANDARD: CAUSE AND EFFECT MATRIX. 2018

MEINKE, K.; NIU, F.; SINDHU, M.A. Learning-Based Software Testing: A Tutorial. **ISoLA 2011 Workshops**, CCIS 336, pp. 200-219, 2012.

MEINKE, K.; SINDHU, M.A. IDS: An Incremental Algorithm for Finite Automata. **School of Computer Science and Communication**, Royal Institute of Technology, Sweden.

MEINKE, K.; SINDHU, M.A.. LBTest: A Learning-based Testing Tool for Reactive Systems. **6th International Conference on Software Testing, Verification and Validation**, p. 447-454, 2013.

PARADKAR, A.; TAI, K. C.; VOUK, M. A.. Specification-based testing using cause-effect graphs. **Annals of Software Engineering**, p. 133-157, 1997.

PNUELI, A. The Temporal Logic of Programs. **Proceeding of the 18 Annual Symposium on Foundations of Computer Science**, p. 46-57, 1977.

PROVOST, J.; ROUSSEL, J. -M; FAURE, J. -M. Generating of Single Input Change Test Sequences for Conformance Test of Programmable Logic Controllers. **IEEE Transactions on Industrial Informatics**, p.1696-1704, 2014.

VEIGA, H. W. Método para teste automatizado de sistemas instrumentados de segurança em plataformas de petróleo. **Dissertação de Mestrado em Engenharia de Automação e Sistemas - PPGEAS**, 2018.

VEIGA, H. W. et al. Automatic Conformance Testing of Safety Instrumented Systems for Offshore Oil Platforms. **Critical Systems: Formal Methods and Automated Verification**, 2017.

WALLACE, D. R.; FUJII, R. U. Software Verification and Validation: Na Overview. **IEEE Software,** p.10-17, 1989.