UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
ENGENHARIA DE CONTROLE E AUTOMAÇÃO

JOÃO VÍTOR ASSMANN

INTERFACE DE SOFTWARE PARA SISTEMAS DE CONTROLE A EVENTOS DISCRETOS

Porto Alegre
2018

JOÃO VÍTOR ASSMANN

INTERFACE DE SOFTWARE PARA SISTEMAS DE CONTROLE A EVENTOS DISCRETOS

Trabalho de Conclusão de Curso (TCC-CCA) apresentado à COMGRAD-CCA da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de *Bacharel em Engenharia de Controle e Automação* .

ORIENTADOR: Prof. Dr. Marcelo Götz

Porto Alegre
2018

JOÃO VÍTOR ASSMANN

INTERFACE DE SOFTWARE PARA SISTEMAS DE CONTROLE A EVENTOS DISCRETOS

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção dos créditos da Disciplina de TCC do curso *Engenharia de Controle e Automação* e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: ____________________________________
Prof. Dr. Marcelo Götz, UFRGS
Doutor pela Universität Paderborn, UPB – Paderborn, Alemanha

Banca Examinadora:

Prof. Dr. João Cesar Netto, UFRGS
Doutor pela Université Catholique de Louvain, UCL – Louvain, Bélgica

Prof. Dr. Ivan Müller, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul, UFRGS – Porto Alegre, Brasil

Prof. Dr. Valner João Brusamarello, UFRGS
Doutor pela Universidade Federal de Santa Catarina – Florianópolis, Brasil

____
Prof. Dr. Mário Roland Sobczyk Sobrinho
Coordenador de curso
Engenharia de Controle e Automação

Porto Alegre, dezembro de 2018.

DEDICATÓRIA

Dedico este trabalho aos meus pais, pelo apoio em todos os momentos dessa jornada.

Dedico o trabalho também aos meus colegas alunos da Escola de Engenharia, em especial aos colegas da Engenharia de Controle e Automação, que se mantêm firmes apesar das dificuldades estruturais da Universidade.

AGRADECIMENTOS

Agradeço em primeiro lugar aos meus pais por tudo que já fizeram por mim.

Agradeço aos amigos Thiago Lehr Companhoni, Bruno Exner, Carlos Miguel Prescendo Tonin, Rafael Pergher, Thomás Schulze e Alef Farah pela ajuda e apoio no trabalho e também pela conversa jogada fora, que ajudou a superar os momentos mais difíceis.

“Brasil acima de tudo,
Deus acima de todos”
Presidente Jair Messias Bolsonaro

RESUMO

Sistemas a Eventos Discretos estão dia após dia mais presentes na nossa vida. Para garantir o funcionamento desses sistemas, uma das teorias que vem sendo usada é a Teoria de Controle Supervisório, que tem se mantido mais restrita a área acadêmica. Nesse trabalho foram então analisadas arquiteturas de Sistemas de Controle a Eventos Discretos e suas implementações na literatura. Foi então proposta e implementada em hardware portátil a modularização de uma dessas arquiteturas, através de uma interface, com o objetivo de facilitar futuros desenvolvimentos na área. Foram apresentados os materiais e métodos utilizados, uma visão global da implementação e seus principais detalhes. Foi feita uma comparação com sistemas já existentes para a validação da implementação, além de um exemplo de como a proposta pode facilitar o trabalho na área. Por fim, foram propostos novos desenvolvimentos tanto na área de Sistemas de Controle a Eventos Discretos como no desenvolvimento dessa interface.

Palavras-chave: Engenharia de Controle e Automação, Sistemas a Eventos Discretos, Controle Supervisório, Interface.

ABSTRACT

Discrete Events Systems are day after day more present in our lives. To ensure their proper functioning, one of the current theories being used is the Supervisory Control Theory, which has remained restricted to the Academia. In this paper, Discrete Events Control Systems architectures and their implementations in the litearature were analyzed. Then, a modularized version of one of these architectures was proposed and implemented in mobile hardware through an interface, with the objective of facilitating future developments in the area. The materials and methods employed were presented as well as an overview of the implementation and its details. A comparison with an already existing system was made to validate the implementation, and also an example of how this proposal can facilitate the work in the area was given. In the end, new developments were proposed, both in the Discrete Events Control Systems domain as well as in the developed interface.

Keywords: Control and Automation Engineering, Discrete Events Systems, Supervisory Control, Interface.

SUMÁRIO

LISTA DE ILUSTRAÇÕES 10

LISTA DE ABREVIATURAS 11

LISTA DE SíMBOLOS 12

1 INTRODUÇÃO 13
1.1 Motivação 13
1.2 Objetivo 14
1.3 Estrutura do Trabalho 15

2 REVISÃO DA LITERATURA 16
2.1 Sistemas a Eventos Discretos 16
2.1.1 Linguagem e Autômatos 17
2.1.2 Teoria de Controle Supervisório 18
2.1.3 Implementação de Controle a Eventos Discretos 20

3 MATERIAIS 22
3.1 Ferramentas de Software e Bibliotecas 22
3.1.1 Escolha da Linguagem de Programação 22
3.1.2 LibFAUDES 22
3.1.3 DESTool 23
3.1.4 FlexFact 23
3.1.5 Outras Ferramentas 23
3.2 Hardware 24
3.2.1 Raspberry Pi 24

4 MÉTODOS 26
4.1 Estrutura do Sistema de Controle 26
4.2 Embarcando a biblioteca libFAUDES 28
4.3 Implementação 29
4.3.1 Leitura do arquivo .dev com a biblioteca TinyXML 30
4.3.2 Mapeamento Eventos-ModBus 30
4.3.3 Interface 32
4.3.4 Supervisório 34

5 ESTUDO DE CASO E RESULTADOS 36
5.1 Caso Estudado 36
5.2 Comparação de Sequências de Eventos 37
5.3 Modificação do Caso Estudado 39

6 CONCLUSÃO E TRABALHOS FUTUROS 41
6.1 Conclusão 41
6.2 Trabalhos Futuros 42

BIBLIOGRAFIA 43

LISTA DE ILUSTRAÇÕES

Figura 1: Autômato e sua representação em *ladder diagram* 14
Figura 2: Autômato representando a linguagem L_1 17
Figura 3: Sistema G controlado pelo supervisório S 19
Figura 4: Diagrama de blocos do Sistema Modular Clássico 20
Figura 5: Exemplo de Sistema Modular Local 20
Figura 6: Janela do software FlexFact com destaque para o log de eventos . . . 24
Figura 7: Raspberry Pi utilizado 25
Figura 8: Estrutura proposta para o Sistema de Controle 26
Figura 9: Estrutura proposta para o Sistema 27
Figura 10: Trecho do arquivo com a relação de eventos e registradores ModBus . 30
Figura 11: Diagrama UML da classe Endereco 31
Figura 12: Diagrama UML da classe Evento 31
Figura 13: Fluxograma de funcionamento da Interface 33
Figura 14: Fluxograma de funcionamento do programa supervisório 34
Figura 15: Planta simulada no software FlexFact 36
Figura 16: Planta simulada em funcionamento 37
Figura 17: Trecho da comparação entre sequências de eventos, com destaque para diferenças entre a ordem executada 38
Figura 18: Planta modificada 39
Figura 19: Planta modificada em funcionamento 40

LISTA DE ABREVIATURAS

SED	Sistema(s) a Eventos Discretos
UML	Unified Modeling Language
RAM	Random Access Memory
XML	eXtensible Markup Language
SO	Sistema Operacional
CLP	Controlador Lógico Programável
TCS	Teoria de Controle Supervisório
API	Application Programming Interface

LISTA DE SÍMBOLOS

X	conjunto de todos os estados de um sistema
E	conjunto de eventos de um sistema (alfabeto);
f	função de transição que mapeia a transição de um estado para outro
Γ	lista de eventos atualmente executáveis
x_0	estado inicial
X_m	conjunto de estados marcados
L_r	linguagem requerida
L_m	linguagem marcada

1 INTRODUÇÃO

O mundo está cada vez mais permeado de equipamentos elétricos e eletrônicos que funcionam a base de comandos discretos. Isso ocorre tanto no meio industrial, com motores, esteiras, distribuidores e até mesmo robôs, através de comandos como liga/desliga ou para frente/para trás. Esses são comandos que muitas vezes implicam em um funcionamento de uma série de equipamentos, sensores e atuadores, como no caso dos robôs, mas que ainda são, essencialmente, comandos únicos, discretos. No meio comercial existem escadas rolantes, portas automáticas, máquinas de vendas, que funcionam também através de comandos discretos que podem vir de sensores ou botões.

Em todos esses casos, para o correto funcionamento desses sistemas, pode ser usado um Sistema de Controle a Eventos Discretos, que modela o sistema de acordo com a premissa de que ele funciona através desses comandos simples, atômicos, e que o submeta a funcionar segundo regras pré-definidas. Pode-se dar como exemplo uma prensa, que quando recebe um comando *'on'* executa a sua função, porém deseja-se estabelecer uma 'regra' para que isso ocorra apenas quando sejam detectadas ambas as mãos do seu operador sobre sensores fora da área de prensa.

Sistemas desse tipo são utilizados para controle de veículos auto guiados (G SILVA; HERING DE QUEIROZ; CURY, J., 2010), parques elétricos (KHARRAZI; MISHRA; SREERAM, 2018) e outros sistemas, porém ainda são escassas. Isso se dá principalmente devido à falta de softwares industriais e padrões reconhecidos amplamente, além de Engenheiros ainda pouco experientes com a modelagem e aplicação de sistemas do tipo (WONHAM; CAI; RUDIE, 2018). Finalmente, então, as implementações atualmente propostas para esses sistemas de controle acabam sendo *hardcoded*, isto é, não permitem facilmente que o controle seja mudado, ou ainda acabam resultando em sistemas tão complexos que precisam ser simplificados para serem implementados, o que nos leva para a motivação desse trabalho.

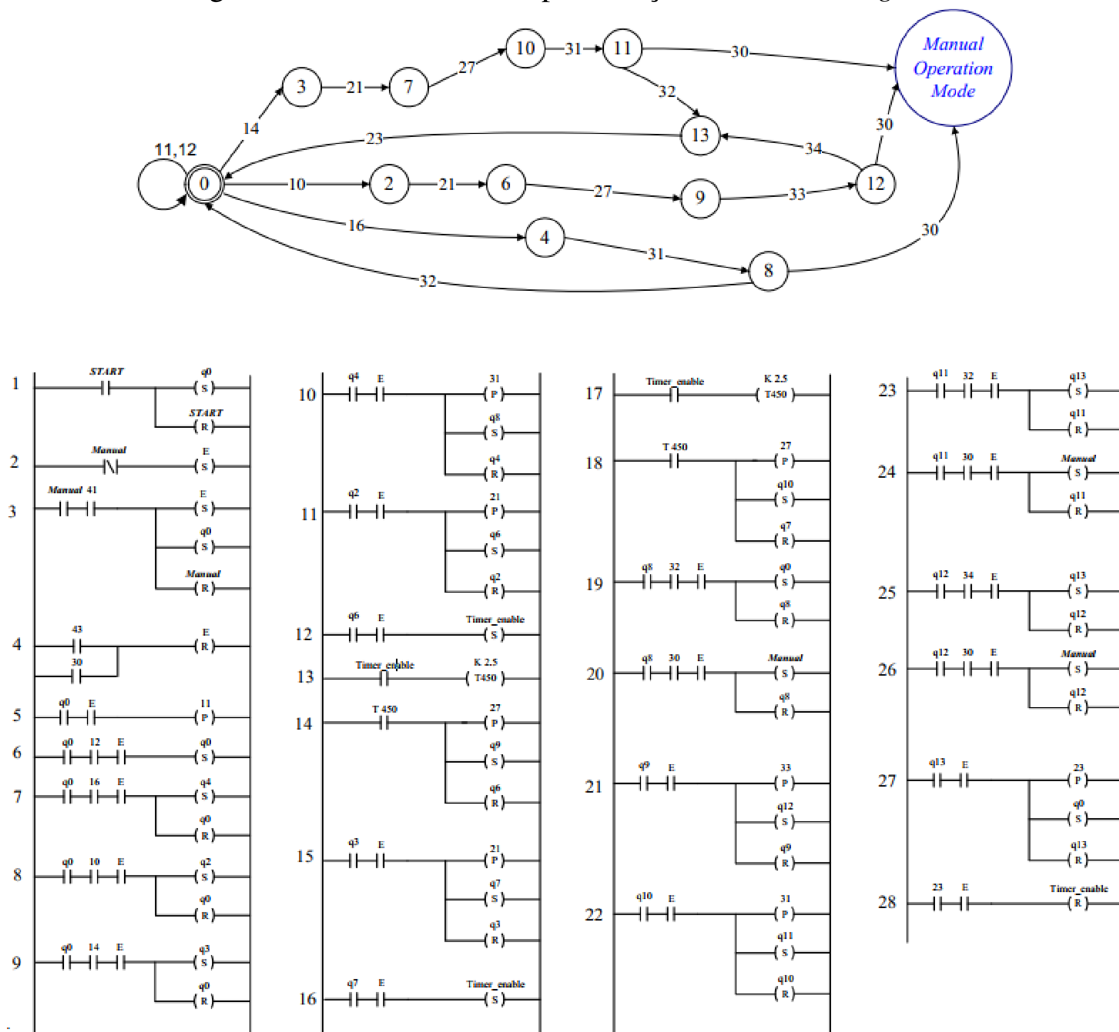
1.1 Motivação

Já existem propostas sólidas no sentido da modularização da geração dos sistemas de controle a eventos discretos que envolvem a parte matemática do cálculo desses sistemas, chamados de supervisórios (QUEIROZ; CURY, J. E. R., 2000). Também encontram-se na literatura propostas de modularização da implementação desses sistemas, que giram em torno da criação automática de código para ferramentas como MATLAB (CURRY et al., 2016) ou código em Java (DRAGERT; DINGEL; RUDIE, 2008), porém elas acabam envolvendo *overhead* desnecessário, como no caso da execução do software MATLAB, que se tornaria inviável em um chão de fábrica.

Em outros artigos (FABIAN; HELLGREN, 1998) (NOORBAKHSH; AFZALIAN, 2009)

existem propostas de implementações de autômatos em linguagem *ladder* para aplicação industrial. No entanto, são necessárias contrapartidas como a redução no número de estados dos autômatos ou ainda, como se pode ver na Figura 1, o código gerado é de difícil leitura e manutenção, pois não há uma relação rápida e clara entre qual registrador corresponde a qual evento/estado. Também nota-se que não existem padrões largamente aceitos pela indústria para essas implementações, sendo que essas acabam sendo feitas caso a caso, o que acaba prejudicando os sistemas como um todo, devido a dificuldade de manutenção decorrente.

Figura 1: Autômato e sua representação em *ladder diagram*



Fonte: (NOORBAKHSH; AFZALIAN, 2009)

1.2 Objetivo

Tendo essas questões em vista, foi proposta nesse trabalho uma modularização não do cálculo dos sistemas de controle em si ou da implementação destes, mas uma modularização da arquitetura dos sistemas de controle por eventos discretos como um todo, tendo como base uma proposta já existente (HERING DE QUEIROZ; CURY, J., 2002). Com isso, é possível melhor desacoplar a leitura e escrita de dados de baixo nível vindos de sensores e causadas por comandos, por exemplo, e se focar num desenvolvimento mais eficiente vi-

sando apenas o supervisório em si. Isso pode contribuir tanto para o avanço na pesquisa da área, como, de uma maneira mais prática, facilitar a ampliação e modificação de sistemas de controles implementados com essa arquitetura. Já existem propostas nesse sentido, como no artigo citado, porém elas acabam sendo implementadas de maneira *hardcoded*, o que torna os sistemas pouco flexíveis. Existe então um ganho para o entendimento e desenvolvimento do sistema com essa teoria, mas não para a flexibilidade do mesmo devido à implementação.

Deseja-se então desenvolver uma arquitetura que possa ser utilizada em mais de um sistema, ou então que possa facilmente comportar alterações no sistema que ela está controlando. A estrutura proposta também pode ser implementada em uma plataforma embarcada, portátil, de maneira que ela possa facilmente ser reutilizada. Uma das vantagens que se obtém com isso é uma maior facilidade na mudança ou ampliação do sistema sendo controlado, sejam essas mudanças efetuadas apenas no supervisório, como no sistema todo.

1.3 Estrutura do Trabalho

No Capítulo 2 será feita uma revisão da literatura, onde serão revisados os conceitos de Sistemas a Eventos Discretos, Linguagem e Autômatos e Teoria do Controle Supervisório. No Capítulo 3 serão descritos os materiais, tanto de software como de hardware utilizados para o desenvolvimento do trabalho. No Capítulo 4 serão descritos os métodos utilizados, como a estrutura proposta e a sua implementação. Já no Capítulo 5 será apresentado um estudo de caso e alguns dos resultados obtidos com o trabalho, e finalmente no Capítulo 6 serão apresentadas as conclusões e possíveis trabalhos futuros.

2 REVISÃO DA LITERATURA

2.1 Sistemas a Eventos Discretos

Inicialmente, é necessário definirmos o que é um Sistema. Segundo (IEEE, 2000), um Sistema é um "Conjunto de elementos que interagem para a realização de uma função não realizável por nenhum dos elementos individuais". Em seguida, podemos dividir os tipos de Sistemas em dois: Estáticos e Dinâmicos. Nos primeiros, as saídas dependem apenas da entrada atual e são geralmente mais simples, como divisores de tensão, somadores, subtratores. Já nos segundos, a saída depende não só da entrada atual, mas também de valores anteriores e de certa forma pode-se dizer que possuem memória. Exemplos de Sistemas Dinâmicos são circuitos Resistor-Capacitor (RC), Resistor-Indutor (RL) ou sistemas mecânicos massa-mola. Sistemas dinâmicos serão alvo de estudo neste trabalho.

A dita 'memória' dos sistemas dinâmicos pode ser representada com 'estados'. A definição de 'estado', segundo (IEEE, 2000), diz que "O estado de um sistema no tempo t_0 é a informação requerida em t_0 tal que a saída $y(t)$ para todo $t \geqslant t_0$, seja univocamente determinada por esta informação (estado) e por um sinal de entrada $u(t)$ para $t \geqslant t_0$". Define-se também como 'espaço de estados' o conjunto de todos os possíveis valores que um estado pode ter.

Na teoria de Sistemas a Eventos Discretos, descrita daqui por diante como SED, portanto, o espaço de estados é naturalmente descrito por um conjunto de valores discretos, como por exemplo: $\{0, 1, 2, ...\}$, denotado por X. A variável independente em SED é o evento. Um evento e pode causar uma transição de um estado para outro do sistema, representar a troca de nível lógico de um sinal (sensor/atuador), representar a alteração dos níveis lógicos de um grupo de sensores/atuadores, ou até algo mais abstrato, como o envio ou recebimento de comandos/mensagens. Os possíveis eventos de um sistema são então modelados por um conjunto também discreto denotado por E e chamado de 'alfabeto' do sistema.

A necessidade do desenvolvimento dessa teoria se deu devido à rápida evolução da comunicação, computação e tecnologias da informação, que são novos sistemas dinâmicos regidos não só por leis físicas, mas também por leis definidas pelo ser humano. Esses sistemas tem dinâmicas regidas por ocorrências assíncronas de eventos discretos, algumas vezes controláveis, como o apertar de um botão, outras vezes não, como o sinal de um sensor.

A base matemática baseada em equações diferenciais que tem sido usada para se modelar e estudar processos e sistemas regidos unicamente pelas leis da natureza, onde a variável independente é o tempo, acaba, então, sendo insuficiente. Dessa maneira, surge a necessidade do desenvolvimento de novas estruturas, técnicas e ferramentas para essa nova geração de sistemas (THIELE L. VANBEVER L., 2018).

Alguns desses novos tipos de sistemas regidos por regras determinadas pelo homem são os sistemas de automação que são utilizados na indústria, comércio e até mesmo residências. Esses sistemas operam motores, elevadores, esteiras, máquinas, através de sinais que representam comandos como ligar e desligar, segundo regras previamente definidas pelo projetista, ou ainda de maneira forçada, com o apertar de um botão. Além disso, eles também recebem informações de eventuais sensores que façam parte do sistema, como por exemplo sensores de detecção de presença.

Podemos, então, dividir os eventos em não-controláveis e controláveis. Os primeiros são aqueles que podem comumente ser associados a sinais vindos de sensores, por exemplo, pois são eventos que não podem ser executados propositalmente pelo sistema de controle. Já os eventos controláveis são aqueles associados a comandos que podem ser dados por um usuário ou pelo sistema de controle, seguindo uma lógica previamente definida, como a ativação de um motor ou esteira.

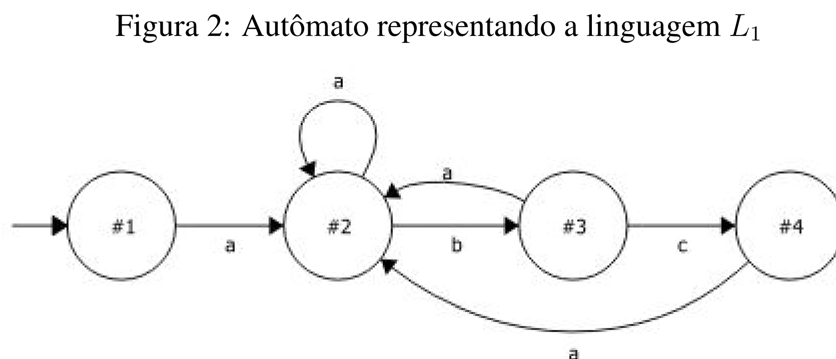
2.1.1 Linguagem e Autômatos

Existem diversas técnicas de modelagem de SED, entre elas destacam-se as linguagens e os autômatos. Podemos então definir uma Linguagem L como o conjunto de todas as sequências finitas de eventos que podem ocorrer em um Sistema a Eventos Discretos. Essas sequências (palavras), naturalmente são compostas por eventos que estão definidos no espaço de estados do sistema.

Por exemplo, em um determinado sistema que possua um conjunto de eventos $E = \{a, b, c\}$ podemos definir uma linguagem L_1 como a linguagem em que o evento a pode ocorrer a qualquer momento, o evento b apenas após o evento a e o evento c apenas após o b. Dessa maneira, teríamos a linguagem $L_1 = \{a, ab, abc, aa, aabc, ...\}$, que seria composta de todas as sequências possíveis dentro das regras impostas.

Para uma melhor visualização e definição das linguagens, visto que elas podem ter tamanho infinito, pode-se utilizar então a teoria de autômatos (CASSANDRAS; LAFORTUNE, 2006). O autômato que representaria a linguagem L_1 definida anteriormente pode ser visto na Figura 2.

Figura 2: Autômato representando a linguagem L_1



Fonte: Autor

Ou seja, iniciando-se no estado #1, só podemos seguir adiante com o evento a, o que nos leva para o estado #2, que nos permite a execução do evento a ou b. Seguindo essa lógica, podemos observar que a sequências de eventos executados nos permitirá formar apenas 'palavras' contidas nas regras definidas para a formação da linguagem L_1.

Diz-se então que um determinado autômato G pode ser definido pela sêxtupla:

$$G = (X, E, f, \Gamma, x_0, X_m),$$

onde:

X é o conjunto de todos os estados;
E é o conjunto de todos os eventos do sistema (seu alfabeto);
f é a função de transição $f : X \times E \rightarrow X$, ou seja, a função que mapeia a transição de um estado para outro;
Γ é a lista de eventos atualmente executáveis;
x_0 é o estado inicial do sistema; e
X_m é o conjunto de estados marcados do sistema.

Os estados marcados são uma decisão de modelagem do sistema. Eles podem representar o estado em que uma tarefa é finalizada, ou então um estado de repouso do sistema que se está descrevendo. Podemos definir, então, a linguagem marcada L_m de um sistema, como a linguagem gerada por todas as sequências de eventos que terminam em algum estado marcado do sistema.

2.1.2 Teoria de Controle Supervisório

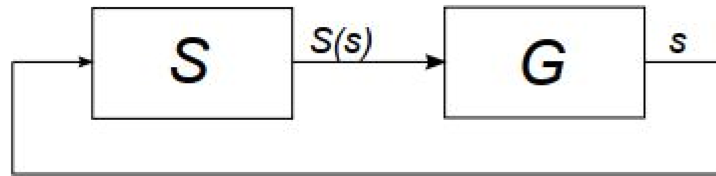
A Teoria de Controle Supervisório para SED foi fundamentada na década de 90 por W. M. Wohnam e P. J. Ramadge (RAMADGE; WONHAM, 1987) e (WONHAM; RAMADGE, 1987), e tem como objetivo impor restrições a um determinado modelo de SED para que este atenda a certas especificações desejadas. Ou seja, deseja-se limitar a linguagem de um determinado SED, modelado por um autômato G, a uma sublinguagem requerida L_r.

Para a realização do controle supervisório, primeiramente se modela o comportamento livre do sistema (chamado na teoria de planta), isto é, considera-se todas as possibilidades físicas de sequências de eventos, sem restrições. Em seguida, modela-se com autômatos cada uma das restrições desejadas ao comportamento livre do sistema e então cria-se um novo sistema pela composição síncrona do autômato original com os autômatos de restrição. Essa composição, então, não permite que um determinado evento passível de ser executado segundo a modelagem do sistema livre G aconteça pois ele está inibido pela restrição. Isso implica então na eliminação da possibilidade de acontecer certas sequências da linguagem livre, objetivando assim a linguagem L_r.

Essa linguagem que é obtida através da composição entre os autômatos da planta e das restrições pode ser, porém, não-controlável, ou seja, seria uma linguagem que para existir, o supervisório teria de impedir um evento não-controlável de ocorrer. Sendo assim, a Teoria de Controle Supervisório propõe uma técnica sobre como calcular um supervisório que gere uma linguagem controlável (RAMADGE; WONHAM, 1987), (WONHAM; RAMADGE, 1987). Essa técnica se baseia em achar dentro da linguagem L_r uma linguagem mais restritiva, porém controlável e que seja também não-bloqueante, isto é, não leva o sistema para um estado em que não hajam transições que permitam sair dele (também chamado de *deadlock*).

Esse novo sistema que gera essa linguagem mais restrita L_r, não-bloqueante e controlável, é chamado de 'supervisório', e é denotado pela letra S. O sistema de malha fechada em que S controla G pode ser visto na Figura 3, onde a planta G envia sua sequência de eventos executados *s* para o supervisório, que em troca envia para a planta o conjunto de eventos habilitados ou desabilitados $S(s)$. O supervisório então age de maneira dinâmica, observando a evolução do sistema G pela análise de todos os eventos executados, sabendo sempre em que estado este se encontra. Em malha fechada a ação de controle é feita, então, desabilitando os eventos que poderiam ser executados pelo sistema livre original G, mas que estão restritos segundo os requisitos do sistema. São executados

Figura 3: Sistema G controlado pelo supervisório S



Fonte: (CASSANDRAS; LAFORTUNE, 2006)

então apenas os eventos controláveis permitidos pelo supervisório e os não-controláveis, que não podem ser impedidos pelo supervisório.

2.1.2.1 Controle Monolítico e Modular

Os primeiros tipos de Sistemas de Controle a Eventos Discretos foram os chamados ‘controles monolíticos’, em que se modelava o sistema a ser controlado e suas restrições levando em conta todos os equipamentos e sistemas que o compunham. Um dos problemas dessa abordagem, entretanto, é o rápido crescimento dos autômatos calculados, e, por consequência, o número de estados. Este aumento implica na dificuldade de seu manuseio (cálculo e manutenção), assim como na sua implementação prática, tornando-a muitas vezes infactível.

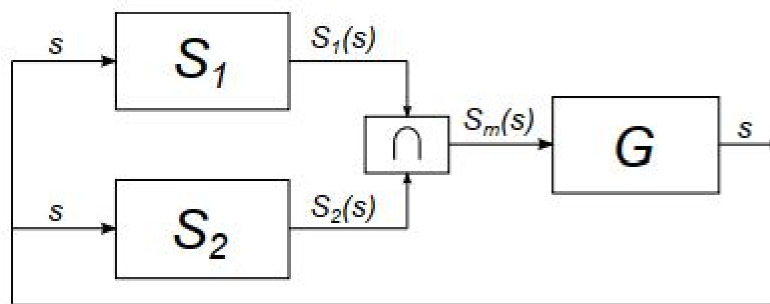
Para diminuir o esforço de cálculo desses autômatos, são propostas então, duas novas arquiteturas de Sistemas de Controle: Modular Clássico e, mais recentemente, Modular Local. O Controle Modular Clássico visa a redução dos cálculos através da ‘quebra’ do Supervisório Monolítico em supervisórios menores, responsáveis por atender a restrições individualmente. Um dos problemas dessa arquitetura, é que apesar de se usar vários supervisórios que tem como objetivo controlar partes independentes do sistema, ainda é necessário se calcular o autômato completo da planta.

Pode-se ver a arquitetura Modular Clássica na Figura 4, em que dois supervisórios S_1 e S_2 observam a sequência de eventos s que ocorre na planta G e então indicam o conjunto de eventos habilitados através de $S_m(s)$, que é composto através da operação de interseção entre os conjuntos de eventos habilitados $S_1(s)$ e $S_2(s)$. Alternativamente os supervisórios poderiam enviar os conjuntos de eventos desabilitados, fazendo com que ao invés de uma intersecção entre os conjuntos, seja necessária uma operação de adição para a construção do conjunto de eventos desabilitados. Os eventos que são executados pelo sistema, então, são aqueles que são permitidos por todos os supervisórios, através da interseção entre os conjuntos de eventos permitidos por cada um deles.

Já nos Sistemas de Controle Modular Local, propõe-se além da “quebra”dos Supervisórios, a divisão também do próprio autômato representativo do sistema (planta) (QUEIROZ; CURY, J. E. R., 2000). Para isso, aproveita-se a arquitetura naturalmente modular das especificações dos sistemas, e evita-se o cálculo do autômato completo do sistema, o que pode gerar uma economia de recursos computacionais.

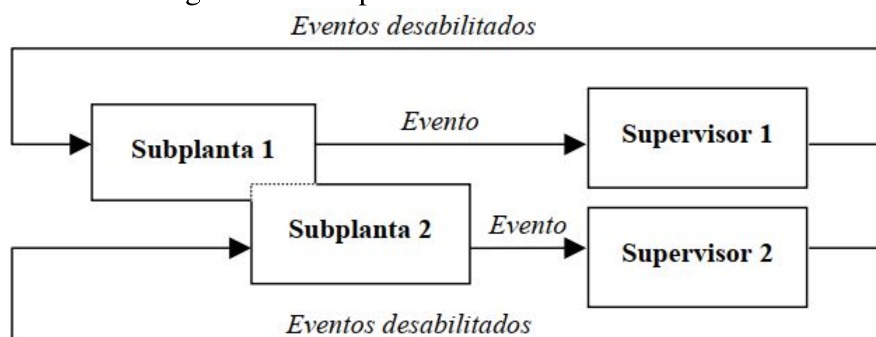
Cria-se, então, o conceito de ‘Planta Local’, ou subplanta, que são subsistemas do sistema completo totalmente assíncronos em termos de eventos, isto é, subsistemas independentes entre si. Nessa arquitetura, cada um dos sistemas supervisórios controla um dos subsistemas independentes, de maneira que os cálculos dos autômatos menores exigem menos esforço computacional do que o cálculo completo do autômato da planta e do supervisório monolítico. Um exemplo desse tipo de arquitetura pode ser visto na Figura 5.

Figura 4: Diagrama de blocos do Sistema Modular Clássico



Fonte: (CASSANDRAS; LAFORTUNE, 2006)

Figura 5: Exemplo de Sistema Modular Local



Fonte: (QUEIROZ, 2000)

2.1.3 Implementação de Controle a Eventos Discretos

A implementação prática de Sistemas de Controle a Eventos Discretos baseados em autômatos possui algumas dificuldades. Uma delas é que a teoria diz que a planta gera de maneira autônoma os eventos, porém isso difere da prática, pois dado um sistema controlado dessa maneira, é necessário de alguma forma se escolher algum dos eventos controláveis disponíveis para serem executados. Além disso, muitas vezes os eventos são gerados em respostas a comandos, e não espontaneamente gerados pela planta. Também, pode-se mencionar que os sistemas computacionais utilizados para se efetuar esse controle são amostrados, isto é, as entradas são amostradas em tempos regulares, e não em todo instante de tempo. Isto culmina no fato de que a detecção de eventos só é possível entre uma amostra e outra, o que pode acarretar que sejam detectados dois ou mais eventos sem se saber qual veio primeiro, o que pode levar o sistema supervisório a um estado que não corresponde ao estado real da planta. Também seria possível que se executassem tantos eventos entre uma amostra e outra de tal maneira que o sistema volte para o estado em que estava antes da segunda amostra, fazendo com que uma sequência de eventos simplesmente não seja detectada. Ainda seria possível que durante a execução do supervisório, seja feita uma escolha de evento para ser executado que deveria na verdade ser bloqueado em seguida devido a um evento ocorrido na planta enquanto é feita essa escolha (FABIAN; HELLGREN, 1998) (WONHAM; CAI; RUDIE, 2018).

Em outros artigos (NOORBAKHSH; AFZALIAN, 2009), (G SILVA; HERING DE QUEIROZ; CURY, J., 2010) a implementação do Sistema de Controle, especialmente da interface entre os equipamentos e seus sinais de baixo nível é implementada de maneira *hardcoded*, ou seja, programada diretamente em linguagens de baixo nível, como C ou

até mesmo *ladder diagram*. Apesar desse tipo de implementação ter um benefício que é um funcionamento mais ágil e determinístico do sistema, uma grande desvantagem é a sua inflexibilidade e impossibilidade de mudança rápida. Ou seja, para situações em que não seja necessária uma grande robustez e que se prevejam mudanças constantes no sistema, pode não ser vantajoso.

3 MATERIAIS

3.1 Ferramentas de Software e Bibliotecas

3.1.1 Escolha da Linguagem de Programação

Duas linguagens foram inicialmente propostas para o desenvolvimento do trabalho: Python e C++. Elas foram escolhidas devido à familiaridade do autor com elas. Cada uma tem suas vantagens e desvantagens.

Python é uma linguagem chamada ‘interpretada’, isto é, ela não gera um código de máquina binário e seus programas podem basicamente ser executados em qualquer plataforma com pouquíssimo esforço, desde que ela tenha instalado e possa executar um interpretador. Python também é uma linguagem de ‘alto-nível’, o que significa que ela abstrai os detalhes do computador e pode permitir um desenvolvimento mais simples e uma maior automação da geração de código (KUHLMAN, 2009).

Já o C++ é uma linguagem ‘compilada’, o que indica que ela gera um binário executável na plataforma onde ela foi compilada. Isso significa que um programa escrito em C++ precisa ser recompilado para cada plataforma onde ele será executado, seja Windows, Linux ou Mac. Uma das vantagens disso é que programas em C++ tendem a ser mais rápidos, pois são constituídos de instruções de baixo nível que executam diretamente no processador. O C++ também pode ser considerado também uma linguagem dita de ‘baixo-nível’, pois dispõe de funções e capacidades de manipulação de memória, apesar de ser bastante utilizada na Programação Orientada a Objeto, uma filosofia de análise e desenvolvimento de software baseada na abstração de detalhes de implementação (STROUSTRUP, 2014).

Levando então em consideração essas vantagens e desvantagens, finalmente foi escolhida para o desenvolvimento do trabalho a linguagem C++. Isso se deu devido a sua relativa rapidez de execução, o que é interessante no ambiente de controle industrial, além da sua fácil aplicação aos conceitos de Programação Orientada a Objeto. Outro fator importante foi a disponibilidade de bibliotecas e recursos que utilizam a linguagem, principalmente a biblioteca libFAUDES, que será descrita na Seção 3.1.2.

3.1.2 LibFAUDES

A biblioteca libFAUDES, para SED, criada e mantida pelo Lehrstuhl für Regelungstechnik (LRT) da Universidade Erlangen-Nürnberg é composta por uma série de classes e funções que implementam algoritmos para autômatos finitos e linguagens regulares (MOOR; SCHMIDT, 2018a). Ela leva em consideração a Teoria de Controle Supervisório introduzida por Ramadge e Wonham e tem como objetivo reduzir o esforço da implementação de métodos para o controle de SED e melhor divulgar para o público a

existência destes métodos.

Esta biblioteca foi escrita em C++, é de código aberto e possui uma API (*Application Programming Interface*) disponível, de forma que não é exigido um grande esforço para o desenvolvimento de novos programas para se trabalhar com SED. As classes e funções mais utilizadas são encapsuladas e disponibilizadas para fácil utilização.

3.1.3 DESTool

O DESTool é uma ferramenta utilizada para a modelagem, análise e composição de SED através de autômatos. É uma ferramenta gráfica que implementa de maneira intuitiva os conceitos presentes na biblioteca libFAUDES, porém foi utilizada nesse trabalho apenas no sentido de criar o supervisório do sistema usado como exemplo e para fins de testes. Igualmente ela é mantida pelo Lehrstuhl für Regelungstechnik (LRT) da Universidade Erlangen-Nürnberg.

3.1.4 FlexFact

O FlexFact é outro software desenvolvido pelo mesmo grupo de trabalho que desenvolveu os softwares libFAUDES e DESTool. Ele é uma ferramenta que tem como propósito a simulação de uma planta modular de manufatura e dos eventos discretos associados a esta. Este software simula esteiras, distribuidores, máquinas de processos, esteiras rotatórias, além de sensores para detecção das peças de trabalho e do estado atual de cada ferramenta.

Cada equipamento possui diversos eventos associados a si. Eles podem ser: (i) não-controláveis, ou seja, são relacionados a sensores de detecção de peças ou fim de curso e não podem ser forçados pelo usuário, ou desabilitados pelo supervisório; ou (ii) controláveis, que são eventos relacionados com o funcionamento desses equipamentos, como ligar ou desligar, que podem ser forçados pelo usuário ou supervisório.

Conforme vão ocorrendo, os eventos podem ser monitorados em tempo de sua execução no sistema, podendo ser visualizados pela janela do programa, como pode ser visto na Figura 6, e podem ser lidos ou forçados por ferramentas externas através do protocolo ModBus, por exemplo. O FlexFact aceita se comunicar em dois protocolos: ModBus TCP/IP ou *"Simplenet"*, um protocolo proprietário. Através da comunicação ModBus, o software troca os sinais dos elementos, permitindo inclusive que ele interaja com um CLP qualquer. Já através do protocolo *"Simplenet"*, as mensagens transmitidas já carregam diretamente qual evento foi executado.

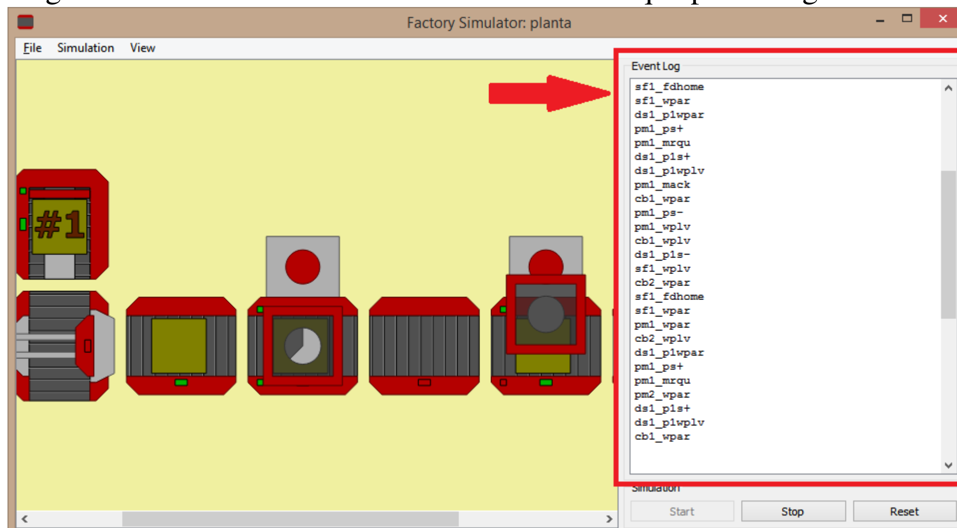
O FlexFact pode, então, exportar as suas informações de comunicação ModBus ou *"Simplenet"*, além das informações sobre os eventos de uma respectiva planta modelada, em um arquivo '.dev'. Esse arquivo disponibiliza essas informações em um formato XML que pode ser facilmente lido por outros softwares.

3.1.5 Outras Ferramentas

3.1.5.1 Microsoft Visual Studio

O Microsoft Visual Studio é uma IDE (*Integrated Developing Environment*) disponibilizada pela Microsoft para o desenvolvimento de código C++. Esta ferramenta é mais apropriada para o desenvolvimento de softwares para plataforma Windows, porém com a extensão VisualGDB é possível compilar e depurar código para plataformas *mobile* como *Android* e *iOS*, além de Linux.

Figura 6: Janela do software FlexFact com destaque para o log de eventos



Fonte: Autor

3.1.5.2 MODBUS++

MODBUS++ é uma biblioteca de código aberto para a abstração de comandos ModBus para C++ e foi utilizada para a comunicação ModBus entre o código executado no RaspberyPi e o sistema simulado da planta modular que envia seus sinais nesse protocolo.

3.1.5.3 TinyXML2

Esta biblioteca foi utilizada para se fazer a leitura dos arquivos .dev disponibilizados pelo FlexFact que indicam qual o endereço e porta do servidor ModBus que será utilizado pela planta simulada e os endereços dos registradores ModBus equivalentes a cada um dos possíveis eventos/sinais controláveis e não-controláveis da planta. O XML é um tipo de linguagem de marcação desenvolvido para ser facilmente lido tanto por humanos quanto por máquinas. A biblioteca TinyXML2, então, disponibiliza diversos métodos em C++ para a leitura de arquivos com informações nesse formato.

3.2 Hardware

3.2.1 Raspberry Pi

A plataforma portátil escolhida foi um Raspberry Pi 3 modelo B, que pode ser visto na Figura 7. Ele tem um processador Quad Core 1.2 GHz Broadcom BCM2837 e 1 GB de memória RAM. O Raspberry Pi é um mini computador em placa única capaz de rodar sistemas operacionais Linux como Ubuntu ou ainda Windows 10 IoT, uma versão reduzida do sistema operacional Windows 10 para sistemas embarcados. No caso desse trabalho, o sistema operacional utilizado é o Raspbian Jessie, uma versão modificada da distribuição Debian de Linux para rodar no Raspberry.

Este equipamento tem conectividade Bluetooth, Ethernet e WiFi, o que se torna bastante interessante para o trabalho, pois permite a sua conexão com uma rede local que possua plantas e sistemas supervisórios rodando sem a necessidade de cabeamento especializado para isso. Também, como quase qualquer sistema Linux, é possível acessar o dispositivo através de conexão SSH a partir de qualquer computador na rede local ou

Figura 7: Raspberry Pi utilizado

Fonte: Autor

conectado à internet. E ainda, como para seu funcionamento é necessária apenas uma alimentação de 5 V, seria possível utilizá-lo sem qualquer conexão física, a partir de um *Power Bank* ou semelhante. Tudo isso contribui para a alta portabilidade do sistema e facilidade de uso.

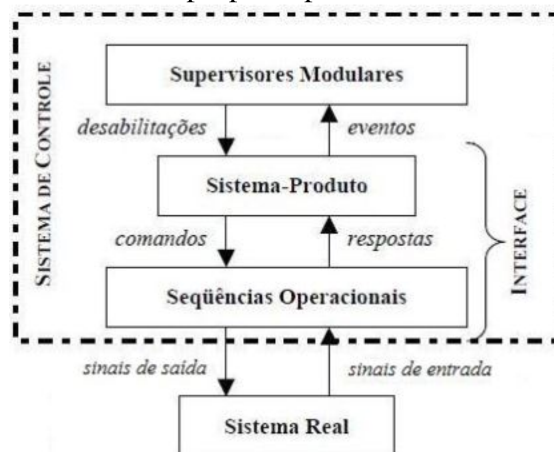
4 MÉTODOS

4.1 Estrutura do Sistema de Controle

(HERING DE QUEIROZ; CURY, J., 2002) propuseram em seu trabalho uma estrutura em três níveis para o Sistema de Controle a Eventos Discretos. Ela pode ser vista na Figura 8. Esses três níveis são: Sequências Operacionais, Sistema-Produto e Supervisores Modulares. Esses níveis tem como objetivo tentar aproximar a Teoria de Controle Supervisório do mundo digital e síncrono do controle industrial. O nível mais acima na figura, são os supervisórios modulares gerados conforme a teoria mencionada na Seção 2.1.2.

Já os outros dois níveis formam o que é chamado de 'interface' do Sistema de Controle, pois são esses os níveis responsáveis por traduzirem sinais de entrada, provenientes do 'Sistema Real' em eventos para o nível 'Supervisores Modulares'. E em sentido inverso, traduzir os eventos que devem ser executados em sinais para a planta real. Esses níveis tentam emular a característica suposta da TCS de que os eventos são gerados espontaneamente pela planta e são então responsáveis por escolher arbitrariamente, ou segundo alguma lógica determinada, quais eventos devem ser executados.

Figura 8: Estrutura proposta para o Sistema de Controle



Fonte: (G SILVA; HERING DE QUEIROZ; CURY, J., 2010)

O nível de Sequências Operacionais trabalha como uma interface entre o sistema real e o nível Sistema-Produto. Essa etapa do processo traduz sinais de baixo nível, como por exemplo vindos de um barramento ModBus, em *respostas* que são repassados para a etapa Sistema-Produto.

Essa segunda etapa, por sua vez, pode ser um modelo teórico da planta real e tem como objetivo traduzir essas *respostas* em *eventos* para os Supervisores. Estes, então, são

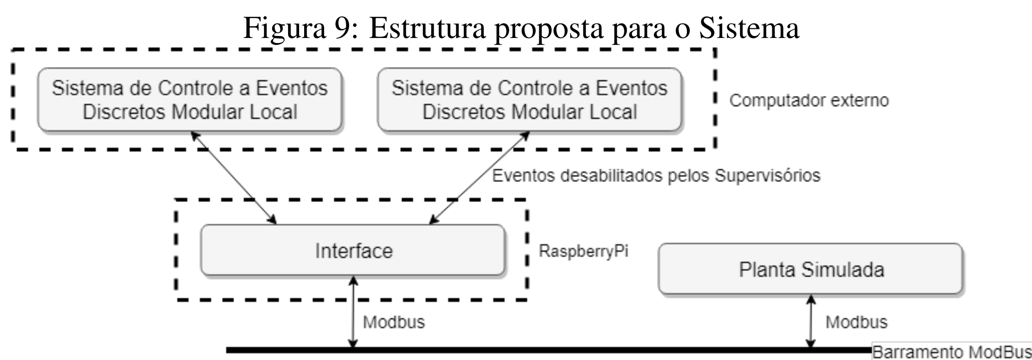
responsáveis por saber em que estado o sistema real se encontra, de maneira a indicar quais eventos controláveis estão desabilitados e retornando essa informação para a etapa anterior.

O nível Sistema-Produto, com essa informação, decide qual evento deve ser executado em seguida, informando a etapa de Sequências Operacionais. Essa última, sabendo quais sinais de baixo nível executam o evento requerido, os envia para o barramento.

Nesse trabalho, então, foi proposto o desenvolvimento de uma interface flexível, que possa ser usada com diferentes sistemas, ao contrário do que foi feito em (G SILVA; HERING DE QUEIROZ; CURY, J., 2010), por exemplo, já que no artigo citado, a interface foi programada diretamente na linguagem C e feita especificamente para aquele caso. Uma das vantagens para uma implementação desse tipo é um bom desempenho do sistema. Porém como desvantagem, caso seja necessária uma mudança, será necessário alterar o código da interface novamente (de forma manual), o que a torna mais propensa a erros.

Visando então uma utilização mais modular e que possibilite de maneira mais fácil que sejam feitas alterações no sistema, propõe-se a estrutura que pode ser vista na Figura 9. Uma planta será simulada no software FlexFact (Seção 3.1.4), se comunicando com a interface via ModBus, enviando informações sobre seus sensores e recebendo sinais que indicam quais ações devem ser tomadas.

Figura 9: Estrutura proposta para o Sistema



Fonte: Autor

A interface, por sua vez, estará sendo executada na plataforma Raspberry Pi, que é um sistema compacto e portátil (Seção 3.2.1) e que estará conectado à rede local via WiFi, monitorando o barramento ModBus. Ela é implementada em um hardware portátil para que seja possível facilitar a reorganização de uma eventual planta que ela esteja controlando, além de facilitar a adição de mais interfaces para se melhor efetuar uma eventual mudança de arquitetura. Isso adiciona flexibilidade ao sistema e reduz custos de cabeamento. Já os supervisórios ficam em um computador separado pois pode se supor que estejam rodando em CLPs ou computadores industriais mais robustos e que tem um local físico fixo na planta.

A interface saberá quais eventos estão relacionados a quais registradores ModBus através de um arquivo XML com extensão .dev (a estrutura e leitura desse arquivo será explorada na Seção 4.3.1) criado pelo FlexFact. Dessa maneira, quando um registrador associado a um evento não-controlável mudar seu valor de zero para um, ou vice-versa, a interface saberá traduzir essa informação em um evento para os Supervisores, e, igualmente, quando tiver de executar um evento controlável, saberá quais registradores deverá alterar.

Já os Supervisores estarão sendo executados em um computador externo, de forma a simular uma estrutura de Sistema de Controle Modulares, e, quando receberem da inter-

face a informação de um evento executado, enviarão de volta quais eventos estão desabilitados na situação atual. Com esse conhecimento, a interface, que terá internamente também uma simulação do modelo do comportamento livre da planta, e, consequentemente, saberá quais eventos estariam em tese habilitados, poderá decidir qual evento pode e deve ser executado em seguida e enviará os sinais apropriados via ModBus.

Vale notar que a estrutura utilizada será equivalente a um controle modular clássico (Seção 2.1.2.1), apesar de não ter sido feito o cálculo do autômato completo da planta. Isso se dá devido ao fato de que os autômatos das subplantas estão sendo executados em paralelo. Uma das consequências disso é que é necessária uma sincronização dos supervisórios, no sentido de que é sempre necessário se saber quais eventos estão desabilitados por todos eles. Caso as subplantas fossem executadas de maneira independente, teríamos uma estrutura equivalente ao controle modular local (Seção 2.1.2.1), em que só precisaríamos dos eventos desabilitados por cada supervisor para executar as transições das respectivas subplantas. Assim, esta estrutura comporta também a estrutura de controle modular local, por ser, conforme visto, mais simples do que a atual proposta (pela ausência da necessidade de sincronização das atuações dos supervisórios).

4.2 Embarcando a biblioteca libFAUDES

A biblioteca libFAUDES (Seção 3.1.2) possui diversos recursos relacionados à simulação e modelagem de SED. Através dela é possível realizar a execução de simulações de sistemas de controle com eventos limitados em tempo, condições de parada, simulações com *hardware-in-the-loop*, isto é, simulações com interação com equipamentos que se comunicam em protocolos conhecidos pela biblioteca, como ModBus, por exemplo, entre outras funcionalidades mais complexas não pertinentes a esse trabalho (MOOR; SCHMIDT, 2018a).

Uma das classes base da biblioteca é a classe *Executor*. Ela é capaz de manter informações sobre o estado atual em que o seu autômato se encontra e quais transições estão atualmente habilitadas ou desabilitadas. Uma das classes derivadas dessa base é a *ProposingExecutor*, que é capaz de executar múltiplos autômatos em paralelo, além de ter funções de leitura e escrita em arquivos de registros (logs). Além disso, essa classe pode propor um evento para ser executado em seguida, baseando-se em prioridades que podem ser configuradas para cada evento, ou então, caso mais de um evento com a mesma prioridade esteja habilitado, ela seleciona um deles aleatoriamente. Essa capacidade acabou não sendo utilizada no trabalho pois como os supervisores são executados separadamente, um deles poderia sugerir a execução de um evento que esteja bloqueado por outro, e, além disso, conforme a arquitetura vista na Seção 4.1, a responsabilidade por escolher qual evento deve ser executado é da interface e não do supervisório em si.

Como visto na Seção 2.1.1, um autômato pode ser representado por uma tupla de seis informações, porém a biblioteca libFAUDES utiliza autômatos compostos de uma tupla de cinco informações: conjunto de estados, conjunto de eventos, transições, estados iniciais e estados marcados. Quando um conjunto de autômatos é gerado no software DESTool, ele pode ser exportado para um arquivo de extensão .sim que contém essas informações necessárias para a criação de um objeto de dados corresponde. Então, em tempo de execução, um objeto da classe *ProposingExecutor* é capaz de ler um arquivo desse formato e através das informações nele contidas, simular a execução de um autômato, sabendo sempre seu estado atual, e quais são as transições que levam desse estado para o(s) próximo(s).

A biblioteca libFAUDES, porém, só é disponibilizada em forma binária/executável para as plataformas Windows e para algumas distribuições Linux. Para a utilização da biblioteca no Raspberry Pi, que possui uma distribuição Linux baseada em Debian, chamada de Raspbian, o download do código-fonte da biblioteca foi necessário, a partir do site do projeto. Em seguida, o download e instalação das dependências do código foram feitos, que são as ferramentas e bibliotecas necessárias para a compilação do código. Apenas então foi possível executar a compilação de toda a biblioteca libFAUDES para a utilização de seus componentes no sistema Linux. É importante ressaltar que a biblioteca deve ser compilada com as mesmas ferramentas que serão utilizadas na compilação do software que a utilizará. Nesse caso, foram utilizadas as ferramentas padrão *GNU-Make* do sistema Linux.

Além disso, como foi escolhida para o desenvolvimento do trabalho a IDE Microsoft Visual Studio (Seção 3.1.5.1), que é utilizada principalmente para o desenvolvimento em sistemas Windows, foi necessária a utilização da extensão VisualGDB para a compilação e depuração do código na plataforma Linux escolhida (SYSPROGS, 2018). Essa extensão permite que seja desenvolvido o código na IDE, no sistema Windows, enquanto que ela mantém uma duplicata de todos os arquivos-fonte no dispositivo Linux, o qual fica responsável pela compilação e execução do programa. Isso foi de grande valia para o trabalho aqui desenvolvido, pois permitiu a integração facilitada entre o desenvolvimento no ambiente Windows (onde o autor possui bom conhecimento) com a compilação e depuração do programa na plataforma Linux.

4.3 Implementação

Para a execução do trabalho, foram escritos dois programas em C++: um deles sendo a interface, que é executada no RaspberryPi, e outro responsável por executar o sistema supervisório desenvolvido no DESTool através das funções de simulação disponibilizadas pelo libFAUDES. Esse segundo programa pode ser executado em múltiplas instâncias, cada uma com um supervisório modular, de forma a extrair o máximo aproveitamento desta arquitetura.

Ambos os programas utilizam objetos da classe *ProposingExecutor* da biblioteca libFAUDES para o controle do estado atual do modelo interno livre da planta (na Interface) e do(s) autômato(s) supervisório(s) no sistema de controle. Objetos dessa classe são capazes de ler a informação sobre autômatos gerados no software DESTool, por exemplo, e exportadas em arquivos .sim para a simulação desses autômatos (MOOR; SCHMIDT, 2018b).

Após a leitura dos autômatos exportados no arquivo .sim, é possível executar eventos que estejam contidos em um ou mais deles através do método *ExecuteEvent*. Assim, o objeto *ProposingExecutor* mantém atualizado o estado de cada um dos autômatos e pode consultar quais eventos estão habilitados e/ou desabilitados em cada estado.

Para a comunicação entre a interface e os sistemas supervisórios foram utilizados *sockets* TCP disponibilizados pelas bibliotecas padrões do Windows e Linux. Já para a comunicação entre a interface e a planta simulada foi utilizado um objeto `modbus` da biblioteca MODBUS++ (Seção 3.1.5.2).

Importante destacar também que houve preocupação na implementação em se tornar o código o mais independente de plataforma quanto possível. Usou-se, por exemplo, classes da biblioteca padrão da linguagem C++, como `std::string`, ao invés das classes disponibilizadas pela IDE Visual Studio através das bibliotecas ATL da Microsoft, como

a equivalente `CString`. Dessa maneira a manutenção do código torna-se mais fácil pois não existem grandes diferenças entre as implementações. A única exceção para isso é o uso dos *sockets* nas duas plataformas, em que é necessário utilizar bibliotecas disponibilizadas pelos sistemas operacionais e não há opção equivalente na linguagem, já que se trata de utilização de recursos provenientes do sistema e não independentes. No caso foi necessário incluir o *header* `WinSock2.h` e *linkar* a biblioteca `Ws2_32.lib` do sistema no Windows e para o Linux é necessário incluir os arquivos `sys/socket.h` e `netinet/in.h` no código-fonte.

4.3.1 Leitura do arquivo .dev com a biblioteca TinyXML

A fim de se relacionar endereços ModBus com eventos gerados pela planta simulada no FlexFact, este exporta um arquivo em formato XML com os dados necessários para isso. O arquivo contém os nomes dos eventos, o tipo deles — *input* para eventos não-controláveis e *output* para controláveis —, além dos registradores associados junto com a informação de se esse evento acontece no sinal de subida, quando o valor muda de 0 para 1, ou de descida, quando o valor muda de 1 para 0. No caso de eventos controláveis, ele indica se o evento deve ser executado *setando* um bit, ou seja, forçando um registrador a ter valor 1, ou *resetando-o*, forçando-o a ter valor 0.

Figura 10: Trecho do arquivo com a relação de eventos e registradores ModBus

```
<Event name="sf1_fdon" iotype="output">
<Actions>
<Set address="31"/>
</Actions>
</Event>
<Event name="cb1_wpar" iotype="input">
<Triggers>
<PositiveEdge address="2"/>
</Triggers>
</Event>
```

Fonte: Autor

Na Figura 10 encontra-se um trecho desse arquivo de extensão .dev, em formato XML. Nela, como exemplo, podemos observar o evento `sf1_fdon`, que é do tipo controlável, e que é executado através do *set* do registrador 31 do servidor ModBus. Também vemos o evento `cb1_wpar`, que é do tipo não-controlável, e que ocorre no sinal de borda de subida do registrador 2 do servidor ModBus.

A biblioteca TinyXML permite, então, a navegação dentro da árvore do arquivo .dev através da iteração entre os elementos `Event` irmãos e o acesso aos seus atributos e 'filhos'.

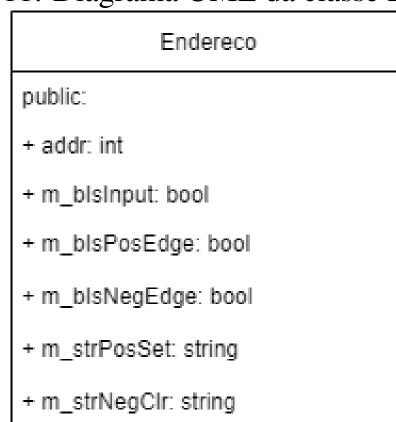
4.3.2 Mapeamento Eventos-ModBus

Através das leituras do barramento ModBus é possível conhecer quais registradores tem seus valores modificados devido aos eventos ocorridos na simulação da planta no FlexFact, porém é necessária uma maneira de se traduzir essa informação. Para isso foram criadas duas classes de objetos para manter as informações sobre endereços de registradores e eventos.

A primeira classe criada é a classe `Endereco`, que pode ser vista na Figura 11, responsável por associar um endereço de registrador ModBus a um ou mais eventos. Mais

de um evento pode estar associado a um registrador ModBus devido ao fato de que um sinal de subida, isto é, uma mudança de valor $0 \rightarrow 1$ pode representar um evento e um sinal de descida ($1 \rightarrow 0$) pode representar outro. Essa classe é usada principalmente para a identificação de ocorrência de eventos não-controláveis, pois relaciona endereços ModBus a eventos.

Figura 11: Diagrama UML da classe Endereco



Fonte: Autor

O principal membro da classe é a variável `addr`, responsável por identificar unicamente esse endereço. O membro `m_bIsInput` identifica se esse registrador é usado para eventos do tipo *input* (não-controlável) ou *output* (controlável). Já os membros `m_bIsPosEdge` e `m_bIsNegEdge` indicam se existem eventos associados a um sinal de subida ou descida, respectivamente. Finalmente, os membros `m_strPosSet` e `m_strNegClr` guardam as *strings* associadas aos eventos de sinal de subida e descida, também respectivamente.

A segunda classe criada, tem o objetivo de fazer a associação contrária: relacionar um evento com um ou mais registradores ModBus. A classe é chamada de `Evento` e pode ser vista na Figura 12.

Figura 12: Diagrama UML da classe Evento

Fonte: Autor

Nessa classe, o principal membro é `m_strEvent`, responsável por identificar unicamente a qual evento cada um dos objetos se refere. Em seguida, tem-se os membros `m_iAddrSet` e `m_iAddrSet2`, que indicam quais registradores ModBus devem ser *setados*, isto é, devem ter seus valores modificados de 0 para 1 quando da ocorrência

desse evento. De maneira análoga os membros `m_iAddrClr` e `m_iAddClr2` indicam quais registradores devem ser *resetados*, ou seja, seus valores devem ser modificados de 1 para 0, quando deve ser executado o evento `m_strEvento`. Esses últimos 4 membros tem um valor padrão -1 pois nem todo evento exige a mudança de 4 registradores ModBus.

Com essas classes, então, foram criados dois mapas globais de tipos `std::map<string, Evento>` e `std::map<int, Endereco>`. O primeiro mapa é utilizado para quando se deve executar um evento, e permite que se encontre rapidamente o objeto `Evento` relacionado para se realizar as ações devidas nos registradores ModBus. Já o segundo é utilizado no sentido contrário, quando se detecta uma mudança em um endereço ModBus e se deseja encontrar rapidamente qual(is) evento(s) está(ão) associado(s) a essa mudança para que seja executado no modelo interno da planta e seja repassado aos supervisórios.

4.3.3 Interface

Na Figura 13 pode-se ver o fluxograma da interface e as etapas de seu funcionamento. A numeração das etapas é usada apenas para referência e não necessariamente indica uma ordem ou prioridade da sua execução.

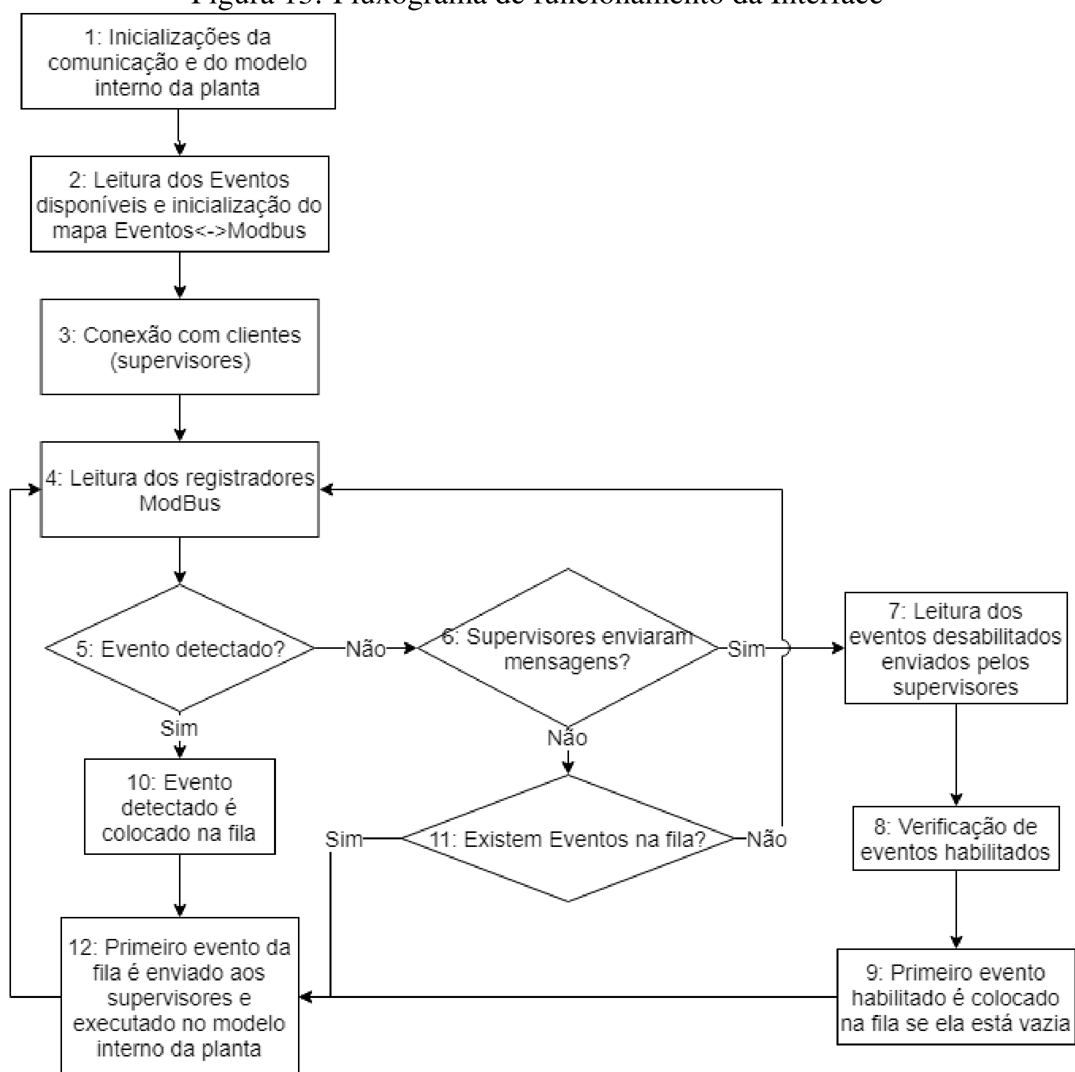
Na etapa 1, são feitas as inicializações necessárias para o funcionamento da comunicação por *sockets* e também são criados os objetos de dados responsáveis por acompanhar o estado atual de toda a planta através do seu modelo completo, que foi previamente modelado no software DESTool. Isso é feito através do objeto *ProposingExecutor*, que é capaz de ler o arquivo .sim exportado e que possui os autômatos representando as subplantas locais.

Na etapa 2, o arquivo .dev gerado pelo FlexFact em formato XML e que contém as relações entre eventos e registradores ModBus é lido e esses dados são armazenados em mapas que relacionam Eventos com Registradores e vice-versa (Seção 4.3.1). Nesta etapa também é gerado o conjunto global de eventos não-controláveis, que será usado na etapa 8. Na etapa 3, são aguardadas as conexões dos clientes, cujo número deve ser previamente determinado, conforme a quantidade de supervisores modulares locais que estarão rodando. Uma vez que eles se inicializaram com sucesso e que a conexão com o servidor ModBus também foi estabelecida, o programa entra no seu laço de funcionamento.

O laço começa com a leitura dos registradores ModBus do servidor, na etapa 4, e o valor atual de todos os registradores é comparado com os valores anteriores. Caso haja alguma diferença, é feita uma consulta ao mapa que relaciona registradores com eventos, para verificar qual evento aconteceu na planta, isso é representado pela etapa 5 do fluxograma. Caso mais de uma diferença tenha sido detectada, todos os eventos correspondentes são colocados em uma fila para execução (etapa 10) e o primeiro evento da fila é executado pelo modelo interno da planta e também é enviado para os supervisores, conforme pode ser visto na etapa 12.

Caso nenhuma mudança tenha sido detectada pelas mensagens do ModBus, é verificado se existe alguma mensagem pendente vinda dos supervisores. As mensagens enviadas por eles contém as transições que estão atualmente desabilitadas e que portanto não podem ser executadas pelo sistema. A interface aguarda a resposta de cada supervisório com a informação dos eventos desabilitados. Ou seja, para seguir adiante no processo, todos os supervisórios devem responder. Assim, a interface calcula de maneira correta a intersecção dos conjuntos de eventos para verificar se algum esteja desabilitado por um ou mais supervisórios (conforme o bloco de intersecção da Figura 4). Isso foi garantido por

Figura 13: Fluxograma de funcionamento da Interface



Fonte: Autor

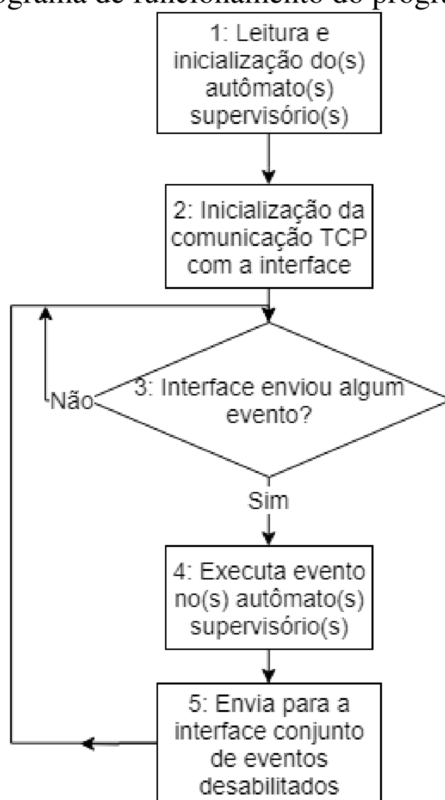
meio do código dos programas através do envio forçado das informações de transições desabilitadas sempre que um supervisório receber uma mensagem da interface, mesmo que seja de um evento não existente em seu alfabeto, e que, portanto, não teria efeito sobre esse supervisório específico.

De posse da informação sobre as transições desabilitadas, é consultado o modelo interno da planta para a verificação dos eventos habilitados por esta. Então através de uma operação de subtração de conjuntos, em que os conjuntos de eventos desabilitados e o conjunto de eventos não-controláveis são subtraídos do conjunto de eventos habilitados, é gerada uma lista dos eventos que são controláveis e atualmente habilitados tanto pelo modelo interno da planta, quanto pelos supervisores modulares locais. Essa lista, em boa parte dos casos, possui nenhum ou apenas um evento e este então é adicionado à fila de eventos a executar. Caso a lista possua mais de uma transição, uma delas é escolhida arbitrariamente para ser adicionada à fila, já que apenas um evento é executado por laço e não se sabe qual o próximo estado em que o sistema vai se encontrar e quais eventos estarão, então, habilitados. A etapa 12 então ocorre, executando-se o evento controlável selecionado no modelo interno da planta, além do seu envio para os supervisores, como ocorre quando um evento não-controlável é detectado. A diferença, nesse caso, é que o mapa que relaciona eventos com registradores ModBus é consultado, e a interface efetua os comandos necessários para a execução do evento na planta simulada.

4.3.4 Supervisório

A Figura 14 apresenta o fluxograma projetado e executado para o supervisório.

Figura 14: Fluxograma de funcionamento do programa supervisório



Fonte: Autor

A estrutura do programa para a execução do sistema supervisório é mais simples do

que a estrutura da interface em si. Assim como na interface, na etapa 1 de inicialização é lido um arquivo .sim que contém os autômatos do sistema de controle modular. Na etapa 2, então, é feita a conexão com a interface através de um *socket* TCP. A partir desse momento o programa do sistema supervisório entra no seu laço de execução.

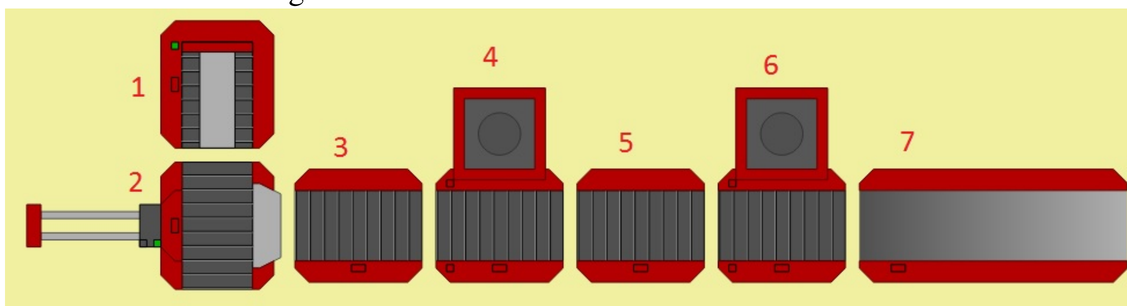
O programa fica então boa parte do seu tempo na etapa 3, aguardando que a interface lhe envie algum evento que possa ter ocorrido de forma espontânea na planta (evento não-controlável) ou que ela (interface) mesmo tenha executado devido ao evento estar atualmente habilitado para isso (evento controlável). Quando é recebida da interface uma mensagem de evento, a transição é então executada nos autômatos supervisórios e através do método *DisabledEvents* do objeto *ProposingExecutor* é recuperado o conjunto de eventos desabilitados em todos eles. Esse conjunto é então passado em forma de *string* para a interface, que então estará na sua etapa 7 de execução.

5 ESTUDO DE CASO E RESULTADOS

5.1 Caso Estudado

Para os testes e validação do funcionamento do sistema, foi necessária a criação de uma planta simulada no software FlexFact. Ela pode ser vista na Figura 15. Essa planta é composta de um alimentador (indicado pelo item 1), um distribuidor (item 2), duas esteiras (itens 3 e 5), duas máquinas de processamento (itens 4 e 6) e um escorregador de saída (item 7).

Figura 15: Planta simulada no software FlexFact



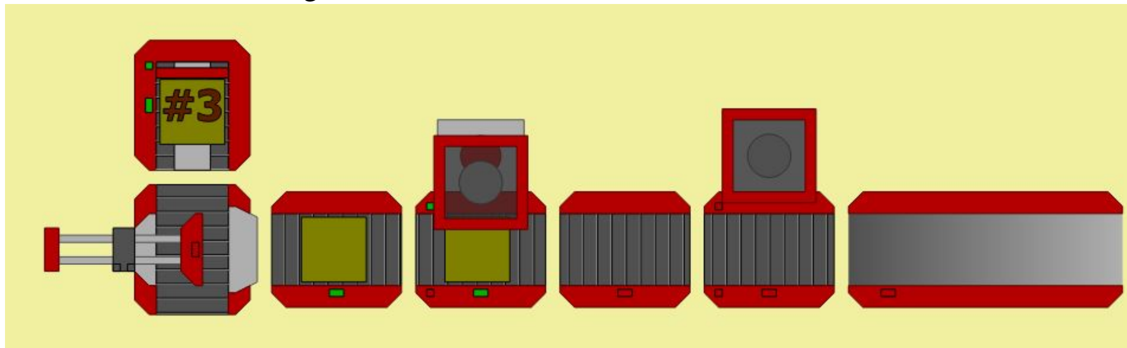
Fonte: Autor

O funcionamento esperado dessa planta é que o usuário que controla a simulação adicione 'caixas' ao alimentador, este então as empurra ao distribuidor, que por sua vez as encaminha para a esteira. Ela em seguida leva estas 'caixas' para a primeira máquina de processamento, que executa um processo arbitrário sobre o item. Findo esse processo, a 'caixa' segue pela segunda esteira até chegar na segunda máquina, onde outro processo é executado. Finalmente, ao fim do segundo processo, a 'caixa' é colocada na esteira de saída, onde fica até o usuário retirá-la.

Nesse caso, algumas das restrições que devem existir são relativas à movimentação das caixas quando os equipamentos na sua frente estão ocupados, isto é, uma esteira, por exemplo, não pode ficar ligada quando uma caixa está sobre si e a máquina à sua frente está ocupada executando um processo. Um exemplo da planta em funcionamento pode ser visto na Figura 16. Nota-se que a primeira esteira está aguardando a finalização do processo na primeira máquina para seguir adiante. Esse tipo de restrição é o que devemos modelar para a criação do supervisório do sistema, pois segundo a modelagem da planta em estado livre, deveria ser possível ligar esta primeira esteira no estado atual do sistema.

Esse sistema foi escolhido para o teste pois inclui diversos tipos de equipamentos disponíveis pelo FlexFact, além de seu sistema supervisório já ter sido testado e comprovado anteriormente na disciplina de Sistemas a Eventos Discretos cursada. Dessa maneira, é

Figura 16: Planta simulada em funcionamento



Fonte: Autor

mais provável que quaisquer eventuais problemas na execução da simulação e dos testes tenham sua origem na implementação da interface e não na modelagem da planta e do seu supervisório.

5.2 Comparação de Sequências de Eventos

Para este trabalho é mais adequado de se realizar uma avaliação qualitativa, e não quantitativa. Assim, o que será proposto nessa seção será uma comparação entre a sequência de eventos executados pelo sistema com os supervisórios modulares locais rodando no software DESTool e a sequência de eventos executados quando se usa a arquitetura proposta, com a interface rodando separadamente dos supervisórios e estes ainda estando separados. Para ambos os sistemas serão aplicadas as mesmas sequências de operações, a fim de se obter as sequências de eventos relacionadas, que tendem a ser idênticas ou equivalentes. Essas sequências indicam o comportamento obtido pelo sistema controlado, ou seja, estamos observando se os comportamentos dos dois são equivalentes. As seguintes operações serão, então, executadas:

1. Serão adicionadas 9 caixas no alimentador
2. Se aguardará que a planta esteja cheia, com todos os equipamentos parados
3. Serão retiradas 4 caixas do escorregador de saída
4. Se aguardará novamente que a planta esteja com todos os equipamentos parados
5. Se retirará 4 caixas do escorregador de saída
6. Quando a última caixa chegar ao fim do escorregador, ela será retirada

Dessa maneira, deve-se iniciar com um sistema vazio com todas as esteiras dos equipamentos paradas e encerrar com o sistema vazio, porém com as esteiras ligadas, aguardando novas caixas, conforme foram projetadas as restrições e os supervisórios. Caso as sequências de eventos sejam iguais ou equivalentes, ou seja, com eventos sequenciais sendo executados em ordem diferente, mas ainda garantindo que as restrições comportamentais impostas sejam atendidas em ambos os casos, pode-se inferir que a arquitetura projetada ao menos garanta um funcionamento equivalente à execução com o software DESTool, que abriga todos os níveis de decisão e execução dos autômatos dentro de si.

Com os *logs* de eventos executados, então, foi usada uma ferramenta de comparação entre textos para se verificar a equivalência entre as sequências executadas. Na Figura 17 pode-se ver um trecho da comparação. No lado esquerdo encontram-se os eventos executados pelo sistema rodando com a interface proposta no trabalho e no lado direito os eventos executados pelo sistema rodando diretamente pelo software DESTool. O que foi observado nessa comparação e que foi destacado na figura, é que os eventos executados permaneceram essencialmente os mesmos, com apenas pequenas diferenças em sua ordem.

Figura 17: Trecho da comparação entre sequências de eventos, com destaque para diferenças entre a ordem executada

32	pm1_mon		
33	ds1_p1wpar	32	ds1_p1wpar
34	ds1_boff	3	ds1_boff
35	ds1_p1m+		ds1_p1m+
		35	pm1_mon
36	ds1_p1s+	36	ds1_p1s+
37	ds1_p1m-		
38	ds1_p1wplv		ds1_p1wplv
		38	ds1_p1m-
39	pm1_mack	39	pm1_mack
40	pm1_pm-	40	pm1_pm-
41	cb1_wpar	41	cb1_wpar
42	cb1_boff	42	cb1_boff
43	pm1_ps-	43	pm1_ps-
44	pm1_bm+		
45	pm1_moff		pm1_moff
		45	pm1_bm+

Fonte: Autor

Pode-se observar na figura, por exemplo, que o evento *'pm1_mon'* ocorreu alguns instantes mais tarde quando o supervisório foi executado via DESTool do que com a interface desenvolvida. Esse evento representa o início do processo da máquina 1, e essa diferença pode-se dar devido a caixa ter sido detectada um pouco antes ou um pouco depois pelo sensor da máquina.

Existe uma diferença também no momento em que ocorre o evento *'ds1_p1m-'*. Ele representa o comando para o retorno do braço que empurra as caixas do distribuidor para a esteira 1. É um comando que deve ocorrer após o braço ter chegado ao seu fim de curso, o que é representado pelo evento *'ds1_p1s+'*, o que ocorre em ambas as situações, com a única diferença sendo que em um caso ele ocorre após o evento *'ds1_p1wplv'* e no outro, antes.

Essas diferenças na ordem de execução de eventos não são um problema pois ainda estão sendo obedecidas as restrições impostas ao sistema. Elas podem se dar devido a diversos fatores, como: (i) o não-determinismo da escolha de eventos pelo DESTool, que quando possui mais de um evento controlável disponível para execução escolhe aleatoriamente entre um dele; (ii) a natureza não-determinística da simulação realizada pelo FlexFact, que pode gerar ordens levemente diferentes de eventos mesmo quando executadas várias vezes a mesma simulação nas mesmas condições; (iii) o fato da comunicação entre a interface e o computador executando a simulação e os supervisórios se dar por protocolo Ethernet, que não é um protocolo para comunicação em tempo real; (iv) flutuações na carga de processamento tanto do computador como do Raspberry Pi, já que ambos estão

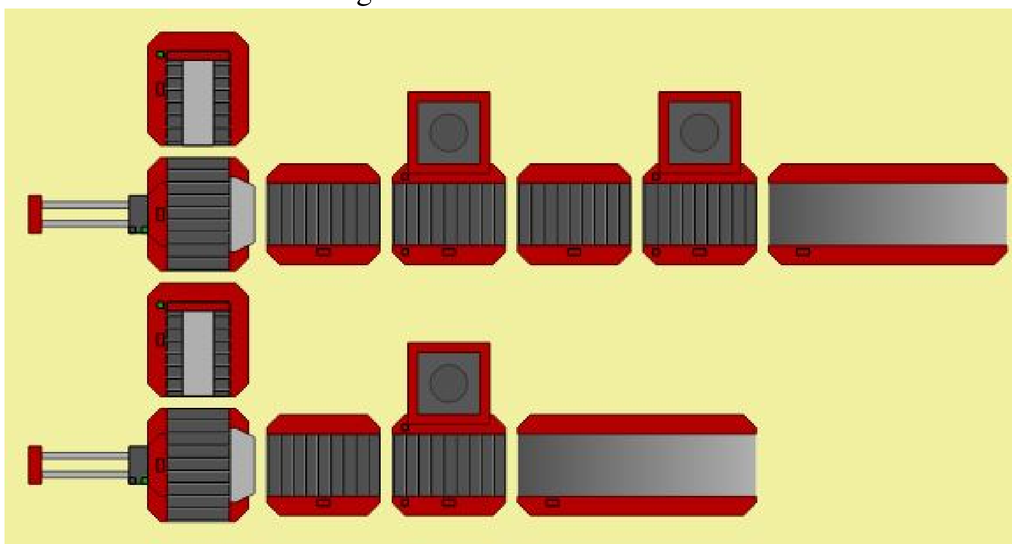
rodando sistemas operacionais que não são de tempo real, e portanto não há uma garantia de determinismo no resultado de operações como essa. Dadas essas considerações, portanto, pode-se concluir que a arquitetura proposta de interface é no mínimo equivalente à execução dos supervisórios diretamente pelo software DESTool.

5.3 Modificação do Caso Estudado

Como um dos objetivos desse trabalho é facilitar a alteração de supervisórios de sistemas controlados por SED, outro teste pertinente para validar a interface é uma modificação na planta simulada utilizada. Em trabalhos como (G SILVA; HERING DE QUEIROZ; CURY, J., 2010) e (HERING DE QUEIROZ; CURY, J., 2002) em que a interface do sistema de controle foi feita de maneira *hardcoded*, isto é, de maneira fixa, a modificação da planta necessita que sejam feitas mudanças manuais não só no sistema de controle, como seria natural, mas também na interface que liga esse sistema ao mundo real.

Será proposta então a modificação do sistema de como ele é visto na Figura 15, para uma estrutura que pode ser vista na Figura 18. Essa nova planta é composta por uma unidade igual à utilizada inicialmente, mais uma unidade menor, com uma máquina e uma esteira a menos. Essa modificação adiciona mais eventos para o alfabeto do sistema, além de ser necessário o controle de uma planta diferente da inicial, impedindo que o mesmo controle seja simplesmente “duplicado”.

Figura 18: Planta modificada



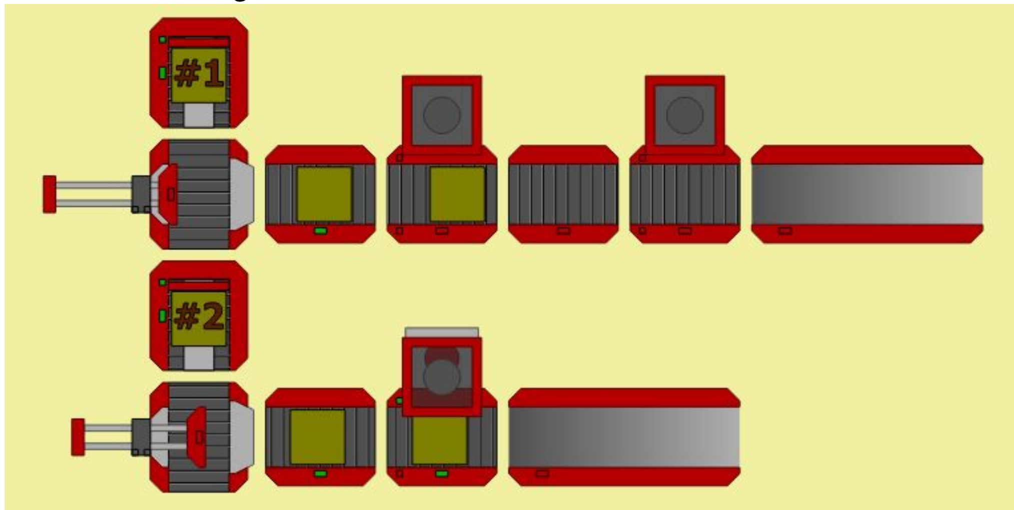
Fonte: Autor

Após executar a mudança no software FlexFact, foi gerado o arquivo .dev de configuração do protocolo ModBus com a relação entre os registradores e os eventos do sistema. Também foi feito um novo controle modular e um novo modelo livre para essa nova unidade, que pode ser considerada uma nova subplanta, o que é relativamente simples de ser feito, dada a natureza modular da modelagem por autômatos, que permite a reutilização dos blocos básicos construídos para o modelo anterior. Realizadas essas etapas, que deveriam ser feitas de qualquer maneira no caso de alteração de uma planta, com a nova arquitetura podemos passar diretamente à execução da interface, carregando o novo arquivo com os modelos internos da planta e executando-se os dois supervisórios anteriores, responsáveis pelo controle da planta antiga e o novo supervisório modular, responsável pela

nova planta.

Na Figura 19 pode-se observar a nova planta simulada funcionando. Destaca-se novamente, portanto, a facilidade introduzida com essa nova arquitetura: não foram necessárias modificações no código da interface para a execução do novo sistema de controle, apenas foram feitas as modificações obrigatórias que deveriam ser realizadas nesse caso, que são a modelagem livre da planta e a criação do seu novo supervisório local.

Figura 19: Planta modificada em funcionamento



Fonte: Autor

6 CONCLUSÃO E TRABALHOS FUTUROS

6.1 Conclusão

Considerando-se o objetivo de implementar uma arquitetura modular para um Sistema de Controle a Eventos Discretos baseado na estrutura proposta por (HERING DE QUEIROZ; CURY, J., 2002), pode-se afirmar que ele foi cumprido. A estrutura do sistema de controle em três camadas: Sequências Operacionais, Sistema-Produto e Supervisores Modulares foi reproduzida com sucesso, onde as duas primeiras representam a interface em software que foi desenvolvida no trabalho.

Na Seção 4.3 mostrou-se que é possível fazer a implementação do sistema em uma plataforma portátil, embarcada, o que representa ganhos na flexibilidade em termos de hardware do sistema. Isso é importante pois é justamente um dos objetivos do trabalho: construir um sistema que possa ser facilmente reutilizado e que permita mudanças na estrutura de controle sem grande esforço manual.

Foi comprovado também que a arquitetura proposta é equivalente a outras já existentes na Seção 5.2. Através da comparação com a execução de um sistema cujo comportamento e funcionamento já se conhecia, foi possível aferir que a nova arquitetura tem capacidade de executar um sistema de controle modular clássico.

Já na Seção 5.3, a interface se comportou como esperado, sem ter dependência de um ou outro sistema supervisório ou planta específica. Isso é bastante importante pois desacopla os supervisórios do sistema real, retirando a necessidade de implementação de uma interface sempre que se desenvolve um novo sistema de controle completo. Assim, outras interfaces genéricas podem ser desenvolvidas, capazes de ‘traduzir’ outros tipos de sinais de baixo nível, transmitidos em outros tipos de protocolos, em eventos. É interessante ressaltar, porém, que nesse trabalho existe uma certa dependência das estruturas fornecidas pela biblioteca libFAUDES, porém em termos de arquitetura, a interface continua válida.

Naturalmente, ainda seria necessária a modelagem do sistema em autômatos e do desenvolvimento do autômato supervisório, e estes devem ter algum formato padronizado que possa ser lido e interpretado pela interface. Nesse sentido, este trabalho mostra que é possível tornar essa arquitetura mais modular e se trabalhar em protocolos e padrões que determinem como um autômato deve ser descrito em um arquivo, como deve ser descrita a relação entre eventos e sinais de baixo nível, entre outros fatores necessários para o funcionamento do sistema de controle.

Além disso, apesar desse trabalho ter usado uma arquitetura que corresponde a um Sistema de Controle Modular Clássico (ver Seção 2.1.2.1), é perfeitamente possível de se utilizar mais de uma interface, com mais de um conjunto de subplantas, em diferentes locais de uma planta, por exemplo, para se atingir uma arquitetura de Sistema de Controle

Modular Local. Também seria simples se efetuar uma mudança na programação da interface para que esta execute mais de um modelo livre de planta, com o mesmo objetivo de se ter uma arquitetura de Controle Modular Local.

6.2 Trabalhos Futuros

Como sugestão para trabalhos futuros, pode-se expandir a interface para ler outros tipos de arquivos não necessariamente dependentes da estrutura fornecida pelo libFAUDES. Ainda nessa mesma linha, pode-se incluir uma interface gráfica, para que o usuário entre manualmente com nomes de eventos e registradores ModBus, o que seria bastante interessante para a expansão de sistemas em que novos eventos surgem, como foi feito na Seção 5.3.

Também seria interessante uma implementação e análise do desempenho tanto da interface quanto do programa supervisório em um SO de tempo real. Isso significaria redução na variabilidade e uma maior repetibilidade dos resultados de controle devido a maior determinismo que esses tipos de sistemas operacionais tem no tempo de processamento de entradas e execução de tarefas, o que se assemelharia a aplicações feitas diretamente em CLPs. Casos de variações cronológicas na ocorrência de eventos como o que é mencionado na Seção 5.2 potencialmente sumiriam ou seriam bastante reduzidos.

Finalmente, pode-se dizer que com essa estrutura é possível também se focar mais no desenvolvimento de cada uma das partes do sistema de controle, e resolver os problemas um a um, mas de maneira mais genérica. Algumas das questões que podem ser trabalhadas na área da Teoria de Controle Supervisório seriam: qual a melhor maneira de ler os sinais de baixo nível? Qual a melhor maneira de se traduzir esses sinais em eventos? Como podemos definir uma linguagem universal de descrição de autômatos e como podemos desenvolver um algoritmo ou sistema para rodar qualquer autômato de maneira genérica? É possível fazer isso de maneira determinística?

Enfim, a área de Sistemas a Eventos Discretos é muito interessante e parece ter várias aplicações no mundo real, mas ainda é necessário um esforço para melhor levar os conceitos desenvolvidos na academia para a indústria. Esse trabalho é apenas mais um de uma série de trabalhos que vem se desenvolvendo ao longo dos anos com o objetivo de se atingir esse maior alcance dos Sistemas de Controle a Eventos Discretos. Observando-se na literatura, através de artigos como (FABIAN; HELLGREN, 1998) e (WONHAM; CAI; RUDIE, 2018) percebemos que há pelo menos 20 anos (se não mais), está se fazendo esse movimento. Os passos são pequenos, mas com a pesquisa não só na própria área da Teoria de Controle Supervisório, como nas áreas de computação e informática, há de se diminuir esse *gap* que existe entre o mundo industrial e o mundo acadêmico.

BIBLIOGRAFIA

CASSANDRAS, C. G.; LAFORTUNE, S. *Introduction to Discrete Event Systems*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387333320.

CURRY, G. L. et al. A modeling language generator for a discrete event simulation language in MATLAB. In: 2016 Winter Simulation Conference (WSC). [S.l.: s.n.], dez. 2016. p. 1013–1023. DOI: 10.1109/WSC.2016.7822161.

DRAGERT, C.; DINGEL, J.; RUDIE, K. Generation of Concurrency Control Code Using Discrete-event Systems Theory. In: PROCEEDINGS of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Atlanta, Georgia: ACM, 2008. (SIGSOFT '08/FSE-16), p. 146–157. Disponível em: <http://doi.acm.org/10.1145/1453101.1453122>. Acesso em: 14 nov. 2018. ISBN 978-1-59593-995-1. DOI: 10.1145/1453101.1453122.

FABIAN, M.; HELLGREN, A. PLC-based implementation of supervisory control for discrete event systems. In: PROCEEDINGS of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171). [S.l.: s.n.], dez. 1998. v. 3, 3305–3310 vol.3. DOI: 10.1109/CDC.1998.758209.

G SILVA, Y.; HERING DE QUEIROZ, M.; CURY, J. Síntese e Implementação de Controle Supervisório Modular Local para um Sistema de AGV, set. 2010.

HERING DE QUEIROZ, M.; CURY, J. Synthesis and Implementation of Local Modular Supervisory Control for a Manufacturing Cell, ago. 2002.

IEEE. *IEEE Std 100-2000: The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition*. [S.l.]: IEEE, 2000. Disponível em: <https://books.google.com.br/books?id=5CV8AQAACAAJ>. Acesso em: 14 nov. 2018.

KHARRAZI, A.; MISHRA, Y.; SREERAM, V. Discrete-Event Systems Supervisory Control for a Custom Power Park. *IEEE Transactions on Smart Grid*, p. 1–1, 2018. ISSN 1949-3053. DOI: 10.1109/TSG.2017.2745491.

KUHLMAN, D. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. [S.l.]: Dave Kuhlman, 2009. Disponível em: <https://books.google.com.br/books?id=mrjOygAACAAJ>. Acesso em: 14 nov. 2018.

MOOR, T.; SCHMIDT, K. *libFAUDES - About*. Disponível em: <https://www.rt.tf.fau.de/FGdes/faudes/>. Acesso em: 14 nov. 2018. Lehrstuhl für Regelungstechnik (LRT) of the University Erlangen-Nürnberg. Out 2018a.

MOOR, T.; SCHMIDT, K. *ProposingExecutor Class Reference*. Disponível em: <https://www.rt.tf.fau.de/FGdes/faudes/csource/classfaudes_1_1ProposingExecutor.html>. Acesso em: 14 nov. 2018. Lehrstuhl für Regelungstechnik (LRT) of the University Erlangen-Nürnberg. Out 2018b.

NOORBAKHSH, M.; AFZALIAN, A. Modeling and synthesis of DES supervisory control for coordinating ULTC and SVC. In: PROCEEDINGS of the American Control Conference. [S.l.: s.n.], jul. 2009. p. 4759–4764.

QUEIROZ, M. H. DE; CURY, J. E. R. Modular control of composed systems. In: PROCEEDINGS of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334). [S.l.: s.n.], jun. 2000. v. 6, 4051–4055 vol.6. DOI: 10.1109/ACC.2000.876983.

QUEIROZ, M. H. DE. *Controle Supervisório Modular de Sistemas de Grande Porte*. 2000. f. 61. Tese (Mestrado em engenharia) – Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina, Florianópolis.

RAMADGE, P.; WONHAM, W. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, v. 25, n. 1, p. 206–230, 1987. Disponível em: <https://doi.org/10.1137/0325013>. Acesso em: 14 nov. 2018. DOI: 10.1137/0325013.

STROUSTRUP, B. *The Essence of C++*. Disponível em: <https://www.youtube.com/watch?v=86xWVb4XIyE>. Acesso em: 14 nov. 2018. University of Edinburgh. 2014.

SYSPROGS. *VisualGDB - Serious cross-platform support for Visual Studio*. Disponível em: <https://visualgdb.com/>. Acesso em: 14 nov. 2018. Sysprogs OÜ. Out 2018.

THIELE L. VANBEVER L., W. R. *Discrete Event Systems*. Disponível em: <https://disco.ethz.ch/courses/hs18/des/>. Acesso em: 14 nov. 2018. Distributed Computing, ETH Zürich. Out 2018.

WONHAM, W.; RAMADGE, P. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, v. 25, n. 3, p. 637–659, 1987. Disponível em: <https://doi.org/10.1137/0325036>. Acesso em: 14 nov. 2018. DOI: 10.1137/0325036.

WONHAM, W.; CAI, K.; RUDIE, K. Supervisory control of discrete-event systems: A brief history. *Annual Reviews in Control*, v. 45, p. 250–256, 2018. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1367578817301876>. Acesso em: 14 nov. 2018. ISSN 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2018.03.002.