Matheus Paiva Loures

Geração Automática de Código para Arduino com base na Teoria de Controle Supervisório de Sistemas a Eventos Discretos

Belo Horizonte, Minas Gerais, Brasil

Junho de 2023

Matheus Paiva Loures

Geração Automática de Código para Arduino com base na Teoria de Controle Supervisório de Sistemas a Eventos Discretos

Projeto de Final de Curso submetido ao Colegiado do curso de Engenharia de Controle e Automação da Universidade Federal de Minas Gerais, como parte dos requisitos necessários à obtenção do título de Engenheiro de Controle e Automação.

Universidade Federal de Minas Gerais – UFMG

Escola de Engenharia

Engenharia de Controle e Automação

Orientador: Professor Lucas Vinícius Ribeiro Alves

Belo Horizonte, Minas Gerais, Brasil

Junho de 2023

Dedico esta monografia à minha querida avó Ieda, à minha tia Aparecida (in memoriam) e a todos que me incentivaram desde o início.

“É preciso durante toda a vida aprender à viver e, o que talvez cause maior admiração , é preciso durante toda a vida aprender à morrer” Sêneca

Resumo

O presente trabalho tem como objetivo principal desenvolver um programa capaz de automatizar o processo de aplicação de sistemas a eventos discretos em sistemas embarcados. Para isso, é utilizada a implementação dos autômatos de estados finitos da biblioteca UltraDES, convertendo-os para código Arduino. Essa abordagem visa facilitar a implementação de soluções de automação em pesquisas acadêmicas realizadas pelo LACSED (Laboratório de Controle e Supervisão de Eventos Discretos).

A metodologia adotada no projeto abrange a seleção cuidadosa da arquitetura, os procedimentos de desenvolvimento e as decisões de projeto relevantes. Cada etapa do processo é minuciosamente descrita, desde a criação dos fluxogramas no planejamento do projeto até a implementação do código correspondente. Essa abordagem fornece uma fonte valiosa de informações para aqueles que desejam desenvolver soluções de automação utilizando a biblioteca UltraDES e a plataforma Arduino, oferecendo uma base sólida e prática para futuros desenvolvimentos nessa área.

Os resultados obtidos demonstram a eficácia da solução desenvolvida. A implementação dos autômatos de estados finitos em código Arduino possibilita a automatização de processos de sistemas a eventos discretos. Além disso, as vantagens da abordagem proposta incluem a facilidade de implementação e manutenção da solução, bem como sua aplicabilidade em pesquisas acadêmicas.

Palavras-chaves: Sistemas a eventos discretos, Teoria de controle supervisório, Modelagem, Programação orientada à objetos e Programação funcional

Abstract

The main objective of this work is to develop a program capable of automating the process of applying discrete event systems to embedded systems. To achieve this, the implementation of finite state automata from the UltraDES library is used, converting them into Arduino code. This approach aims to facilitate the implementation of automation solutions in academic research conducted by LACSED (Laboratory of Control and Supervision of Discrete Events).

The methodology adopted in the project encompasses careful architecture selection, development procedures, and relevant design decisions. Each step of the process is meticulously described, from creating flowcharts in project planning to implementing the corresponding code. This approach provides a valuable source of information for those wishing to develop automation solutions using the UltraDES library and the Arduino platform, offering a solid and practical foundation for future developments in this field.

The obtained results demonstrate the effectiveness of the developed solution. The implementation of finite state automata in Arduino code enables the automation of discrete event system processes. Moreover, the advantages of the proposed approach include the ease of implementation and maintenance of the solution, as well as its applicability in academic research.

Key-words: Discrete event systems, Supervisory control theory, Modeling, Object-oriented programming, Functional programming

Agradecimentos

Este trabalho é fruto de uma jornada que não apenas moldou um engenheiro, mas também lapidou o ser humano que hoje sou. Neste momento, transbordo de gratidão ao Senhor Deus pela dádiva milagrosa da vida, pela oportunidade de estudar em uma das mais notáveis universidades do país e pela chama viva da esperança que Sua graça incendeia em meu coração.

Com reverência e profunda admiração, expresso minha gratidão à Universidade Federal do Estado de Minas Gerais, cujos pilares do saber inflamaram em minha alma a busca incansável por desvendar os segredos do mundo, e que gentilmente me concedeu as ferramentas necessárias para adentrar incansavelmente na busca pelo conhecimento.

Minha gratidão se expande também em direção à minha família, que com apoio incondicional, entrelaçou-se de maneiras inúmeras nessa caminhada, encorajando-me a ultrapassar fronteiras. Agradeço imensamente a Jorge Luiz Loures Martins, Aurea Conceição Paiva Loures, Gabriel Rezon Paiva Loures, Marianna Victória Paiva Loures e Thalyta Ávila Costa, agradeço do fundo do meu coração por todo o amor, dedicação e acolhimento imensuráveis.

Aos amigos que se tornaram minha força e sustento ao longo dessa trajetória sublime - Emanuela de Matos, Caio Conti, Maria Luiza Alves, Gabriel Pereira Duarte e Marcelo Braga - expresso minha gratidão profunda. Vossa amizade e companheirismo foram essenciais em cada etapa desta jornada, presentes em todos os momentos de apoio mútuo e colaboração sincera.

Sou grato ao meu orientador, Professor Lucas Vinícius Ribeiro Alves, cujo suporte inestimável, orientação sábia e ensinamentos valiosos foram a bússola que iluminou o desenvolvimento deste trabalho. Sua experiência e sabedoria foram alicerce para o meu crescimento acadêmico. Agradeço também ao Professor Diógenes Cecilio da Silva Junio, cuja participação na banca trouxe uma perspectiva singular para o trabalho, acrescentando profundamente na busca por um resultado verdadeiramente satisfatório.

Por fim, minha gratidão se estende a todos os membros do Laboratório de Análise e Controle de Sistemas a Eventos Discretos, que generosamente compartilharam sua expertise e contribuíram para minha formação. Em especial, expresso minha gratidão à professora Patrícia Pena, cujos conselhos inestimáveis e orientações preciosas seguirão comigo por toda a minha vida.

A cada um de vocês, envio meu mais sincero e profundo agradecimento. Sem o apoio de cada indivíduo mencionado acima, este trabalho não teria atingido o patamar que alcançou. Vocês são parte indissociável desta conquista e sou imensamente grato por ter tido a honra de contar com sua presença em minha extraordinária jornada.

Lista de ilustrações

Lista de Algoritmos

*

Lista de abreviaturas e siglas

LACSED	Laboratório de Analise e Controle de Sistemas à Eventos Discretos
UFMG	Universidade Federal de Minas Gerais
SED	Sistema a Evento Discreto
TCS	Teoria de Controle Supervisório
AFD	Autômatos Finitos Determinísticos
PDES	Simulação de Eventos Discretos Paralelos
CLP	Controladores Lógicos Programáveis
FPGA	*Field-Programmable Gate Arrays*
SRA	Sistema de Representação de Processo
POU	Program Organization Units
FB	Function Blocks
IDE	*Integrated Development Environment*
IoT	*Internet of Things*

Lista de símbolos

Σ	Conjunto de eventos ou Alfabeto
Σ^*	Conjunto de todas as cadeias finitas sobre Σ
Σ_u	Conjunto de eventos não controláveis
Σ_c	Conjunto de eventos controláveis
L	Linguagem gerada
L_m	Linguagem marcada
G	Autômato da planta global
Q	Conjunto de estados do autômato G
Q_m	Conjunto de estados marcados do autômato G;
δ	Função de transição de estados
$\in$	Pertence
$\subseteq$	Está contindo ou igual a
s	Cadeia
Σ	Sequência de evento
S	Comportamento desejado de G
ε	Cadeia vazia
q_0	Estado inicial
E	Especificação
$\forall$	Para todo

Sumário

1 Introdução

Com base nas ferramentas modernas de produção e nas novas demandas do mercado, surgiu a necessidade de desenvolver formas diferentes de fabricar bens. Assim, é perceptível que a automação de processos avançou significativamente nas últimas décadas, tendo os robôs como protagonistas nos ambientes fabris. Esses novos sistemas de produção não apenas buscam qualidade e redução de custos, mas também enfatizam a importância de personalizar a produção de maneira eficiente, otimizando e automatizando o processo. Nesse contexto, o aumento da produtividade é um desafio constante para a indústria, levando pesquisadores a estudar e aprimorar técnicas de otimização da produção, que visam minimizar o tempo de produção, maximizar lucros e aumentar a segurança dos trabalhadores.

Essas demandas atuais por otimização, personalização e automação nas novas metodologias de produção impõem desafios às estrategias abordadas pelo controle clássico em que se trabalha a modelagem usando equações diferencias e equações de diferença. Isso se deve ao fato desses sistemas interagirem com o ambiente de forma distinta daqueles que são modelados pela forma clássica, percebendo o ambiente por meio de eventos ou estímulos instantâneos. Os sistemas que evoluem pela ocorrência de estímulos instantâneos são conhecidos como Sistemas a Eventos Discretos (SEDs), esses sistemas estão presente nas mais diversas aplicações contemporâneas, incluindo automação de manufatura, redes de computadores, logística de cadeia de produção e robótica.

A compreensão aprofundada desses sistemas e o desenvolvimento de abordagens adequadas são cruciais para atender as demandas de desempenho, eficiência e personalização das práticas de produção modernas e por essa importância o Laboratório de Analise e Controle de Sistemas a Eventos Discretos (LACSED, Figura 1) se apresenta como um grupo de pesquisa da Universidade Federal de Minas Gerais (UFMG) com o objetivo de pesquisar e desenvolver aplicações de SEDs desde 2012.

Figura 1 – Logo do Laboratório de Analise e Controle de Sistemas a Eventos Discretos

1.1 Motivação

O LACSED tem se dedicado, nos últimos anos, a diversas linhas de pesquisa no campo da Teoria de Controle Supervisório. Muitos desses estudos demandam a simulação tanto por meio de *softwares* quanto a implementação desses estudos em plantas físicas didáticas, visando uma melhor visualização dos resultados obtidos. Como parte desses esforços, o laboratório desenvolveu a biblioteca UltraDES, que oferece uma ampla gama de algoritmos e estruturas de dados utilizados na Teoria do Controle Supervisório, incluindo grafos e redes de Petri.

No entanto, no que diz respeito à implementação de SEDs em plantas físicas existe um desafio devido a necessidade de criar um novo programa para cada autômato desenvolvido o que leva a um impacto no tempo necessário para o processo de implementação nas soluções de SEDs desenvolvidas pelo LACSED. Para superar essa limitação, surge a necessidade de desenvolver uma aplicação capaz de converter os algoritmos e modelos de autômatos finitos da biblioteca UltraDES em programas em *C++*, compatíveis com a plataforma *Arduino*.

A solução proposta traria inúmeros benefícios, simplificando e agilizando o processo de implementação física, além de permitir uma melhor visualização e validação dos resultados obtidos. A criação dessa ferramenta proporcionaria uma solução eficiente e prática para o teste dos modelos teóricos desenvolvidos pelo LACSED.

1.2 Objetivos

O objetivo do presente trabalho é desenvolver um programa que converta automaticamente a implementação feita dos autômatos de estados finitos da biblioteca UltraDES para o código *Arduino*. Isso permitirá a aplicação da Teoria de Controle Supervisório de Sistemas a Eventos Discretos nessa plataforma de *hardware*. Além disso, espera-se realizar um estudo de caso simples com um conjunto de esteiras. Dessa forma, os objetivos específicos do projeto são os seguintes:

- Superar o gargalo existente nas implementações físicas de soluções de SEDs desenvolvidas no LACSED, economizando tempo e recursos no processo de implementação.
- Possibilitar uma melhor visualização dos resultados obtidos e contribuir para a compreensão e validação dos modelos teóricos.
- Facilitar a implementação da Teoria de Controle Supervisório em sistemas embarcados, contribuindo para o avanço das pesquisas na área.
- Simplificar a aplicação de Sistemas a Eventos Discretos na plataforma Arduino, permitindo que usuários sem conhecimento avançado em *C++* possam desenvolver a lógica dos autômatos.

1.3 Organização do Texto

O trabalho está organizado da seguinte forma: após a introdução, que apresentou um panorama sobre o assunto, o Capítulo 2 abordará os conceitos fundamentais necessários para o desenvolvimento do projeto. Serão explorados os temas principais de Sistemas a Eventos Discretos, Teoria de Controle Supervisório, bem como uma introdução à plataforma Arduino, que será utilizada na implementação do projeto. Além disso, será apresentada a ferramenta UltraDES, amplamente utilizada pelo LACSED.

No Capitulo 3, serão discutidas as soluções já desenvolvidas para a implementação da Teoria de Controle Supervisório em outros sistemas dedicados, como o CLP e o FPGA. Ademais será enfatizado o que é relevante para a implementação do sistema proposto para o Arduino.

No Capitulo 4, serão apresentados o planejamento do projeto, discutindo a arquitetura escolhida para o desenvolvimento e as escolhas de projeto que foram desenhadas com a finalidade de produzir uma solução robusta.

Já no Capitulo 5 serão apresentados o desenvolvimento originado da metodologia desenvolvida e os resultados obtidos. Por fim, no Capitulo 6 , serão apresentadas as conclusões do trabalho, juntamente com ideias de desenvolvimento para futuros trabalhos relacionados ao tema.

2 Fundamentação Teórica do Projeto

Neste capítulo serão apresentados os fundamentos teóricos que embasam o desenvolvimento do projeto, abrangendo os conceitos básicos de sistemas a eventos discretos, teoria de controle supervisório e a biblioteca UltraDES. Embora essa apresentação não seja exaustiva, ela é essencial para situar o projeto em seu contexto e fornecer uma base sólida para sua implementação.

2.1 Sistemas a Eventos Discretos

Os sistemas dinâmicos podem ser definidos como um conjunto de componentes, dispositivos ou subsistemas conectados, cujos os estados evoluem seguindo um conjunto bem definido de regras. Essas regras descrevem como as variáveis do sistema mudam ao longo do tempo, como interagem umas com as outras e como o sistema se relaciona com o meio através do seus sinais de entrada e saída. Dentro do ramo de estudos de sistemas dinâmicos, existe uma classe desses sistemas que evoluem de acordo com a ocorrência abrupta, em intervalos irregulares, de eventos físicos (CASSANDRAS; LAFORTUNE, 2010).

Essa classe de sistemas é conhecida como Sistemas a Eventos Discretos, os quais podem ser descritos por um conjunto de estados e as transições entre esses estados, que ocorrem em momentos específicos no tempo. Essas transições entre estados estão relacionadas a eventos que podem ser identificados como a ocorrência de um fenômeno espontâneo no sistema, uma ação específica realizada sobre o sistema ou o resultado de um conjunto de condições que são atendidas simultaneamente.

Existem várias técnicas disponíveis para modelar Sistemas a Eventos Discretos, dependendo do aspecto que se deseja analisar. Portanto, SEDs podem ser representados de várias maneiras, utilizando linguagens e autômatos (SKOLDSTAM; AKESSON; FABIAN, 2007), redes de Petri (RU; HADJICOSTIS, 2009) e outras formas de representação. É importante destacar que, independentemente da técnica escolhida para modelar um SED, o momento exato em que um evento ocorre não é tão relevante em comparação com a ordem em que os eventos acontecem no sistema. Além disso, para esse tipo de sistemas, a ocorrência simultânea de dois ou mais eventos não é permitida.

2.1.1 Linguagem

Em um Sistema de Eventos Discretos os eventos podem ser representados como símbolos distintos pertencentes a um conjunto chamado de alfabeto, denotado por Σ. Já as sequências finitas de eventos podem ser interpretadas como palavras em uma linguagem que utilizam dos

símbolos do alfabeto. Dessa forma, o conjunto Σ é o alfabeto e o conjunto Σ^* representa todas as cadeias finitas que podem ser geradas a partir desse alfabeto.

Na prática, é comum ter interesse em um subconjunto específico das sequências compostas por símbolos de um alfabeto. Esse subconjunto é conhecido como linguagem. Uma forma concisa de representar certas linguagens é por meio de expressões regulares, que permitem representar um número infinito de cadeias através de uma expressão fechada. Uma linguagem que pode ser representada por expressões regulares é chamada de linguagem regular. As expressões regulares são compostas por cadeias e operações aplicadas a essas cadeias. As operações válidas em uma expressão regular incluem:

- União: A operação de união de duas linguagens é denotada pelo símbolo $+$ e representa a operação lógica OU;
- Concatenação: A operação de concatenação é representada pela justaposição de duas linguagens, o que significa que elas são colocadas uma após a outra, em sequência;
- Fechamento Kleene: A operação de fechamento Kleene é representada pelo símbolo $*$ e indica que uma determinada cadeia pode ocorrer repetidamente um número arbitrário de vezes;

É interessante observar que o comportamento sequencial de um SED pode ser descrito por meio de um par de linguagens L e L_m sobre o conjunto de símbolos que representam os eventos que afetam o sistema. A linguagem gerada L descreve o conjunto das cadeias fisicamente possíveis de ocorrerem no sistema, ou seja, as sequências de eventos que respeitam as restrições físicas e lógicas do sistema. Enquanto a linguagem marcada L_m descreve o conjunto de cadeias de L que correspondem a tarefas completas, ou seja, as sequências de eventos que levam o sistema a um estado desejado ou aceitável. As linguagens gerada e marcada são fundamentais para a modelagem de SEDs, pois permitem especificar os requisitos e os objetivos do sistema, bem como verificar se o sistema atende a esses requisitos e objetivos (CASSANDRAS; LAFORTUNE, 2010).

2.1.2 Autômatos Finitos Determinísticos

Expressões regulares podem ser usadas para representar todas as linguagens regulares, as quais, por sua vez, podem ser geradas por autômatos. Um autômato é um modelo matemático de uma máquina que reconhece uma linguagem regular através de estados e transições entre eles, na perspectiva matemática é um grafo onde os vértices representam os estados do sistema, as arestas representam as transições, com essa estrutura é possível descrever um sistema dinâmico mapeando o comportamento do mesmo em relação à evolução dos seus estados. Um autômato finito determinístico é definido por uma quíntupla (CASSANDRAS; LAFORTUNE, 2010):

$$G = (Q, \Sigma, \delta, q_0, Q_m) \tag{2.1}$$

Em que

- Q é o conjunto de estados do autômato G;
- Σ é o conjunto de eventos do autômato G, também chamado de alfabeto de G;
- δ é a função de transição de estados da forma $\delta : Q \times \Sigma \to Q$, em que as entradas são um estado e um evento e obtém como saída o próximo estado;
- $q_0 \in Q$ é o estado inicial;
- $Q_m \subseteq Q$ é o conjunto de estados marcados de G;

A função de transição δ pode ser estendida para cadeia de eventos com a função $\hat{\delta}$: $Q \times \Sigma^* \to Q$. Que mapeia para qual estado de Q o sistema transita a partir do estado q com a ocorrência de uma sequencia $s \in \Sigma^*$.

A importância dos autômatos finitos para a representação de SED é, juntamente com as linguagens regulares, verificar se uma cadeia pertence ou não à linguagem, bem como gerar todas as cadeias possíveis da linguagem. Além disso, os autômatos finitos são úteis para modelar sistemas que possuem um número limitado de estados e que respondem a eventos discretos.

A representação mais comum de autômatos é feita através de diagramas de transição. Nessa representação, os estados são representados por círculos e os eventos são representados por setas. O estado inicial é indicado por uma seta extra que aponta para o esse estado. Os estados marcados são representados por dois círculos concêntricos.

Os diagramas de transição são úteis para visualizar o funcionamento do autômato e seguir o caminho percorrido por uma cadeia de entrada. Um exemplo de automato encontra-se na Figura 2:

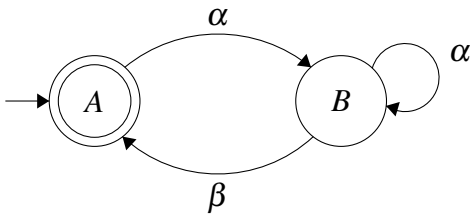

Figura 2 – Exemplo de Autômato

Do automato da Figura 2 pode-se desprender a seguinte quíntupla:

$$G = (Q, \Sigma, \delta, q_0, Q_m) \tag{2.2}$$

em que, para esse autômato, tem-se:

- $Q = \{A, B\}$;
- $\Sigma = \{\alpha, \beta\}$;
- δ:

$$\delta(q, \sigma) = \begin{cases} A & se \quad q = B, \sigma = \beta \\ B & se \quad (q = B, \sigma = \alpha) \; ou \; (q = A, \sigma = \beta) \end{cases}$$

- $q_0 = A$;
- $Q_m = A$;

Existem várias operações que podem ser aplicadas a autômatos, como a Parte Acessível, Parte Coacessível, *Trim* e a composição de autômatos. Essas operações permitem modificar, simplificar ou combinar autômatos para obter novos, de modo que esses reconheçam linguagens diferentes ou equivalentes.

- **Parte Acessível**: consiste no subautômato formado por todos os estados que podem ser alcançados a partir do estado inicial. Essa operação elimina os estados inúteis que nunca são visitados pelo autômato.
- **Parte Coacessível**: consiste no subautômato composto apenas pelos estados que podem alcançar um estado marcado. Essa operação elimina os estados inúteis que nunca levam a um estado de interesse.
- **Trim**: é o resultado da aplicação das operações de parte acessível e parte coacessível sobre um autômato. Essa operação elimina todos os estados inúteis do autômato e obtém um autômato equivalente mais simples.
- **Composição de autômatos**: permite a representação do comportamento conjunto dos autômatos originais. Existem diferentes formas de compor autômatos, como a união, a interseção e a concatenação. É um operação representada pelo operador: $\|$.

2.2 Teoria de Controle Supervisório

A Teoria de Controle Supervisório (TCS) (WONHAM; CAI, 2017), fundamentada na teoria dos Autômatos Finitos Determinísticos (AFD), oferece um método formal para a modelagem de sistemas complexos, conhecidos como plantas, cujo comportamento pode não seguir um padrão bem definido. Essa falta de previsibilidade pode resultar em bloqueios indesejados ou danos ao sistema. Com o intuito de lidar com essas questões, a TCS propõe o desenvolvimento de um agente de controle capaz de garantir a segurança e evitar bloqueios nesses sistemas.

Considerando a perspectiva da linguagem gerada por um sistema modelado por um autômato G, representada por $L(G)$, observa-se a existência de cadeias de eventos inaceitáveis devido à violação de condições de segurança ou restrições de bloqueio desejadas (CASSANDRAS; LAFORTUNE, 2010). Essas cadeias podem ser indesejáveis devido a estados de bloqueio ou estados fisicamente inválidos que possam causar danos ao sistema. Além disso, certas cadeias em $L(G)$ podem conter *substrings* não permitidas, violando a ordem desejada dos eventos, como a concessão de solicitações para uso de um recurso comum que devem ser concedidas de acordo com a ordem de chegada. Portanto, é necessário considerar sublinguagens de $L(G)$ que representem o comportamento admissível para o sistema controlado.

Dessa forma o objetivo do agente de controle, conhecido como supervisor, é garantir um comportamento seguro e não bloqueante em sistemas de malha fechada. O supervisor atua desabilitando e/ou habilitando certos eventos para assegurar um comportamento seguro e não bloqueante no sistema. O desafio central abordado pela teoria de controle supervisório é encontrar um supervisor que atenda às especificações do sistema (E) de forma menos restritiva possível.

Assim, ao considerar a representação do autômato $G = (Q, \Sigma, \delta, q_0, Q_m)$, pode-se observar que $\Sigma = \Sigma_c \cup \Sigma_u$, em que Σ_c representa os eventos controláveis que podem ser identificados como os estímulos externos do sistemas e por esse perfil podem ser alvo de ação de controle, enquanto, Σ_u engloba os eventos não controláveis que são uma interpretação dos fenômenos espontâneos do sistemas e por essa característica não podem ser desabilitados pelo supervisor. Desse modo o supervisor trabalha habilitando e desabilitando apenas os eventos de Σ_c, com o objetivo de fazer com que a planta assuma o comportamento desejado K, em que $K = L(G \| E)$, ou seja K é uma linguagem que é encontra por meio da composição entre a linguagem de da planta (G) e a linguagem das especificações (E). A condição necessária e suficiente para a existência de um supervisor não bloqueante S é que K seja controlável em relação a $L(G)$ e Σ_u.

Essa propriedade garante que o supervisor que implementa a linguagem K não tentará desabilitar eventos não controláveis na planta. Caso o supervisor não seja controlável, é necessário calcular a máxima sublinguagem controlável contida em K na linguagem desejada, denotada por $SupC(K, G)$.

Assim, o controle supervisório tem como objetivo controlar a planta de modo que, ao

adicionar uma estrutura de controle, seja possível variar a linguagem gerada pelo sistema controlado dentro de um limite específico. Assim essa técnica de controle pode ser representada através de uma linguagem chamada de especificação e com ela é obtido o comportamento desejado para o sistema em malha fechada.

2.2.1 Controle Monolítica

Nessa abordagem de TCS é necessário que haja uma única planta, seja porque o sistema foi completamente modelado ou porque foi calculado ao compor várias subplantas modeladas pelos autômatos finitos determinísticos (AFD) G_k, resultando no autômato G estruturado e não vazio. Controlar essa planta implica em adicionar uma estrutura de controle que permita variar a linguagem gerada pelo sistema controlado dentro de certos limites.

O procedimento para projetar um supervisor pela técnica monolítica pode ser resumido pelo método a seguir (PENA, 2007):

1. Modelagem das subplantas e especificações que compõem o sistema;
2. Obtenção da planta global G pela composição das subplantas G_i,do seguinte modo: $G = G_1 \| G_2 \| ... \| G_i$;
3. Obtenção da especificação monolítica E pela composição das especificações E_j ,do seguinte modo: $E = E_1 \| E_2 \| ... \| E_j$;
4. Obtenção da linguagem desejada K para o sistema em malha fechada pela composição de E e $L_m(G)$;
5. Obtenção do supervisor S, cuja linguagem $L(S)$ é a máxima sublinguagem controlável contida em K, de modo que $L(S) \subseteq L(K)$;

2.2.2 Controle Modular

O controle modular, proposto em (WONHAM; RAMADGE, 1988), é uma extensão do controle monolítico desenvolvido para a TCS, com o intuito de facilitar a síntese de supervisores e aumentar a flexibilidade do sistema, ao contrário da abordagem do controle monolítico, em que apenas um supervisor é projetado para restringir o comportamento da planta de forma a satisfazer as especificações em malha fechada, o controle modular clássico divide a tarefa de garantir o cumprimento das restrições em subtarefas, que são atribuídas a supervisores distintos.

A premissa básica desse modelo de controle é que a combinação das ações dos supervisores soluciona o problema original, garantindo-se que a ação conjunta desabilite eventos que sejam desabilitados por pelo menos um supervisor. O teste de modularidade consiste em verificar se todo prefixo de uma cadeia marcada nos supervisores também é prefixo de pelo menos uma cadeia da interseção das linguagens dos supervisores.

O procedimento para projetar os supervisores, seguindo a abordagem modular, é resumido da seguinte maneira (PENA, 2007):

1. Modelagem das subplantas e especificações que compõem o sistema;
2. Obtenção da planta global G pela composição das subplantas G_i,do seguinte modo: $G = G_1 \| G_2 \| ... \| G_i$;
3. Obtenção das linguagens desejadas K_j , $K_j = Ej || L(G)$;
4. Obtenção dos supervisores que implementam as máximas linguagens controláveis contidas em $K_j, S_j = SupC(K_j, L(G)), \forall j$;
5. Teste da modularidade entre os supervisores obtidos;

2.2.3 Controle Modular Local

Uma extensão da técnica modular vista na seção anterior é o controle modular local (QUEIROZ; CURY, 2000) que expande o uso da propriedade modular não apenas nas especificações de controle, mas também na estrutura modular da planta. O objetivo é evitar a explosão de estados que pode ocorrer durante a síntese de um supervisor monolítico, especialmente em problemas de maior escala.

No controle modular local, um supervisor é criado para cada especificação de controle, seguindo uma abordagem semelhante à modular. No entanto, apenas os subsistemas afetados por cada especificação são considerados no cálculo de cada supervisor modular. Dessa forma, cada supervisor local tem uma visão parcial da planta, o que permite uma abordagem descentralizada do controle. Cada supervisor local é controlável e não bloqueante por construção, garantindo um comportamento adequado em relação às especificações locais. No entanto, é importante ressaltar que o comportamento combinado de todos os supervisores locais pode levar a uma situação de bloqueio, em que nenhum evento é habilitado para ocorrer. Portanto, é necessário garantir a coordenação adequada entre os supervisores locais para evitar essas situações.

Uma vantagem significativa de representar as especificações em termos de sistemas locais é que, em alguns casos especiais, é possível identificar a modularidade sem a necessidade de cálculos adicionais. Isso ocorre quando as especificações locais, compostas pelos subsistemas afetados, não compartilham eventos em comum. Essa identificação prévia da modularidade simplifica o projeto e a implementação do sistema de controle, reduzindo a complexidade computacional.

A abordagem modular local continua sendo uma alternativa computacionalmente mais eficiente em comparação com as abordagens modular e monolítica para lidar com sistemas complexos. Ela oferece flexibilidade ao verificar a modularidade sem a necessidade de composição

completa do sistema e permite o controle descentralizado, levando em consideração apenas os subsistemas relevantes para cada especificação.

2.3 UltraDES

O UltraDES (ALVES; MARTINS; PENA, 2017) é uma biblioteca desenvolvida em linguagem C# e baseada no *.NET Framework*, que oferece estruturas de dados e algoritmos para modelagem, análise e controle de SEDs compatível com qualquer linguagem de programação e plataforma que suporte *.NET Standard* 2.0, incluindo ambientes móveis como Android e *iOS*. A biblioteca surgiu como um projeto de pesquisa do LACSED da Universidade Federal de Minas Gerais, com o objetivo de facilitar o desenvolvimento e a validação de métodos e técnicas para o tratamento de Sistemas de Eventos Discretos.

Figura 3 – Logo da biblioteca UltraDES

A biblioteca UltraDES utiliza os paradigmas de programação orientada a objetos e programação funcional. Ela oferece estruturas para representar eventos, estados e autômatos, permitindo a modelagem e análise de sistemas a eventos discretos. Além disso, a biblioteca possui implementações de vários algoritmos, especialmente relevantes na Teoria do Controle Supervisório, além de algoritmos e estruturas de dados para grafos e redes de Petri.

Essa biblioteca pode ser aplicada em diversas áreas práticas que envolvem sistemas de eventos discretos, como controle de processos industriais, planejamento e otimização de rotas, diagnóstico e detecção de falhas, verificação formal de propriedades lógicas e síntese automática de controladores, entre outras aplicações.

Para representar estados, a biblioteca oferece a classe "*State*", que permite ao usuário definir o nome (*alias*) do estado e se ele é marcado ou não. Da mesma forma, para representar eventos, utiliza-se a classe "*Event*" permitindo definir seu nome (*alias*) e se é controlável ou não. A classe "*Transition*" é utilizada para representar uma transição, que recebe como parâmetros um estado de origem, um evento e um estado de destino. Essas estruturas são utilizadas para definir o comportamento dos sistemas de eventos discretos.

Por fim é possível representa um autômato determinístico através da classe "*DeterministicFiniteAutomaton*" que representa os autômatos finitos determinísticos. Essa classe oferece várias operações, como Parte Acessível, Parte Coacessível e Trim, além de fornecer funções para expressões regulares e síntese de supervisores monolíticos e modulares.

No armazenamento das informações de um autômato, a classe utiliza o estado inicial, vetores de eventos e estados, e uma matriz de adjacência. Cada linha da matriz de adjacência contém uma lista ordenada de pares (Evento, Estado), representando as transições de cada estado.

2.4 Arduino

O Arduino (TEAM, August 14, 2020) é uma plataforma eletrônica de código aberto baseada em hardware e software que tem se mostrado uma ferramenta poderosa na implementação física de sistemas em uma ampla variedade de aplicações. Sua popularidade crescente deve-se principalmente à sua acessibilidade financeira, facilidade de uso e versatilidade.

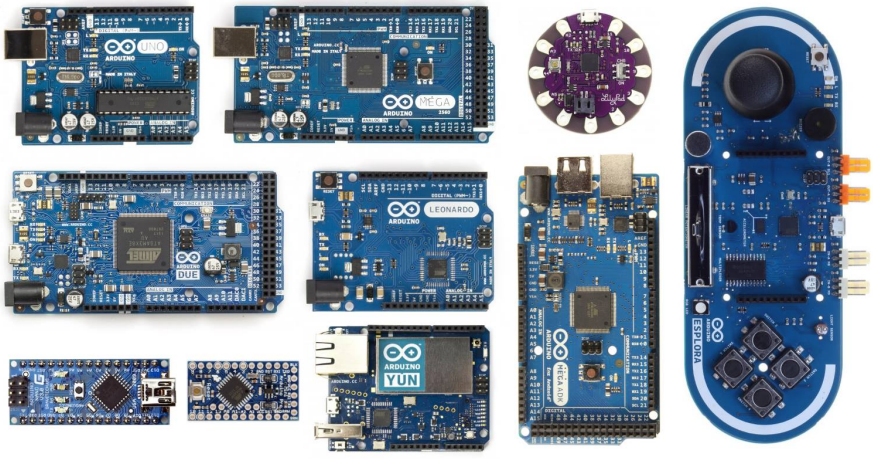

Figura 4 – Exemplos das placas mais comuns

Ele foi projetado para ser uma ferramenta amigável para aqueles que não possuem um conhecimento prévio em eletrônica ou programação. A linguagem de programação utilizada pelo Arduino é baseada no *Wiring* (BARRAGáN, August 14, 2020) e é bastante simplificada, o que facilita o aprendizado e a escrita de código. A IDE do Arduino fornece uma interface intuitiva que permite aos usuários escrever, compilar e carregar seu código para a placa Arduino com facilidade.

Além disso uma característica importante do Arduino é sua versatilidade tendo em vista a existência de diferentes modelos de placas Arduino, cada uma com características específicas que se adéquam a diferentes necessidades. Por exemplo, há placas Arduino projetadas para aplicações em Internet das Coisas (*IoT*), *wearables*, impressão 3D e ambientes embarcados. Essa diversidade de opções permite que os usuários escolham a placa Arduino mais adequada para o seu projeto, levando em consideração fatores como recursos, tamanho e conectividade.

Em suma, o Arduino é uma ferramenta acessível, fácil de usar e versátil que tem simplificado a implementação física de sistemas. Sua combinação de baixo custo, facilidade de

programação e comunidade ativa torna o Arduino uma escolha popular entre estudantes, entusiastas e profissionais que desejam criar e *prototipar* sistemas físicos. Com o Arduino, a implementação de projetos eletrônicos e a transformação de ideias em realidade se tornaram mais acessíveis e emocionantes do que nunca.

2.5 Máscara de Bits

A máscara de bits é um conceito fundamental na área da ciência da computação e redes de computadores. Ela desempenha um papel essencial na manipulação e controle de valores específicos dentro de um dado. Em termos gerais, um bit é uma unidade básica de informação que pode assumir dois estados distintos, representados pelos valores binários 0 ou 1. A máscara de bits, por sua vez, consiste em uma sequência ordenada de bits que é aplicada a um valor binário com o propósito de selecionar ou modificar partes específicas desse valor.

No âmbito da ciência da computação, a máscara de bits funciona como um filtro, permitindo a passagem de determinados bits e bloqueando o acesso a outros, conforme sua configuração estabelecida. É possível utilizar operações lógicas *bitwise*, tais como *and* (e), *or* (ou), *xor* (ou exclusivo), *not* (negação), *shift left* (deslocamento à esquerda) e *shift right* (deslocamento à direita).

Em algumas linguagens de programação a máscara de bits é representada como um tipo inteiro sem sinal, já em outras linguagens, a máscara de bits pode ser implementada como um tipo enumerado, proporcionando a atribuição de nomes significativos aos valores presentes na máscara. Cabe ressaltar que cada abordagem possui vantagens e desvantagens em termos de legibilidade, portabilidade e eficiência.

As aplicações da máscara de bits são vastas em diferentes áreas da computação, abrangendo programação, sistemas operacionais e redes de computadores. Um exemplo comum de utilização de máscaras de bits ocorre na configuração de endereços IP. O endereço IP, que é um número utilizado para identificar um dispositivo na internet, é dividido em duas partes distintas através do emprego de uma máscara de bits. A primeira parte, denominada "parte da rede", indica a qual rede o dispositivo pertence, já a segunda parte, chamada de “parte do *host*”, identifica o dispositivo específico dentro daquela rede. A máscara de bits estabelece o número de bits destinados a cada uma dessas partes, permitindo que os dispositivos de rede encaminhem adequadamente pacotes de dados para a rede correta. Os pacotes de dados, por sua vez, são blocos de informações transmitidos pela internet.

3 Revisão da Literatura

No capítulo anterior, foram apresentados os fundamentos teóricos, a biblioteca Ultra-DES e a plataforma Arduino, os quais são elementos essenciais para o desenvolvimento do projeto em questão. Neste capítulo, serão exploradas soluções já implementadas em sistemas dedicados, com o objetivo de analisar abordagens semelhantes aplicáveis ao sistema proposto para o Arduino. Serão apresentados estudos e projetos que aplicaram a teoria de controle supervisório em outros sistemas, como Controladores Lógicos Programáveis (CLPs), Field-Programmable Gate Arrays (FPGAs) e microcontroladores.

É importante ressaltar que este capítulo não se trata de uma análise comparativa entre as plataformas, mas sim de uma análise dos paradigmas desenvolvidos e de como aplicar essas estruturas na conversão de autômatos finitos para a plataforma Arduino. As técnicas de implementação já desenvolvidas na literatura são apresentadas com o intuito de refinar a solução proposta.

A área de Sistemas a Eventos Discretos e teoria de controle supervisório em sistemas embarcados, juntamente com as pesquisas relacionadas a microcontroladores, CLPs e FPGAs, é um campo em constante evolução, amplamente estudado e aplicado em diversos setores da indústria, como manufatura, automação e controle de processos. Os estudos apresentados neste capítulo são relevantes para o desenvolvimento do projeto proposto para o Arduino, pois permitem explorar abordagens similares já implementadas em outros sistemas dedicados.

Nesse sentido, será enfatizado o que é relevante para a implementação da solução no Arduino, levando em consideração as particularidades desse hardware. Essa etapa é fundamental para garantir que a aplicação desenvolvida seja eficiente e adequada às especificidades do Arduino, considerando a constante evolução da área. Destaca-se, assim, a importância de manter-se atualizado e buscar sempre as melhores soluções para o desenvolvimento de projetos nessa área.

3.1 Controladores Lógicos Programáveis

Entre os trabalhos e projetos que buscam aplicar as técnicas de modelagem de SED e a teoria de controle supervisório, destaca-se a implementação em CLPs. Um exemplo de implementação é o desenvolvido no (FABIAN; HELLGREN, 1998).

Inicialmente, a planta é modelada utilizando autômatos finitos e, com esse modelo em mãos, é realizada a implementação em linguagem de programação *ladder* do comportamento de uma máquina de estados finitos. A forma escolhida para representar a planta como uma máquina de estados em *ladder* é atribuir variáveis booleanas internas a cada estado e evento,

utilizando uma lógica de *AND* booleano para representar as transições entre os estados.

Nesse contexto o trabalho passa a tratar de algumas dificuldades encontradas nesse processo de implementação de SED em um CLP, uma delas refere-se à natureza assíncrona dos sistemas dirigidos a eventos. Ao associar eventos com bordas de subida ou descida de determinados sinais, pode ocorrer o efeito avalanche na implementação, em que o programa pula um número arbitrário de estados durante o mesmo ciclo de varredura. Esse comportamento é resultado da avaliação sequencial das expressões booleanas dentro do CLP, e pode levar a resultados indesejados e comprometer o correto funcionamento do sistema. A abordagem indicada pelo trabalho para lidar com essa dificuldade é classificar os degraus do diagrama *ladder* em uma ordem "inteligente", embora ainda seja um desafio em aberto encontrar a melhor ordem para evitar o efeito avalanche.

Outro desafio encontrado está relacionado à transição de um mundo baseado em eventos para um mundo baseado em sinais. Enquanto eventos são geralmente considerados como não ocorrendo simultaneamente a outros eventos, os sinais podem se sobrepor no tempo. A associação de eventos com bordas de sinal permitiria assumir a não sobreposição de eventos, desde que a detecção da borda de subida pudesse ser garantida. No entanto, devido à execução cíclica do CLP, essa garantia não é possível, exigindo uma cuidadosa abordagem para sincronização e processamento dos sinais, a fim de evitar conflitos e resultados incorretos. Uma solução para essa dificuldade é a utilização de um supervisor insensível ao intervalo captura, em que a decisão de controle não depende da ordem de eventos que ocorrem no mesmo intervalo de tempo. Essas dificuldades ressaltam a importância de explorar e desenvolver abordagens eficientes e confiáveis para a implementação de Sistemas a Eventos Discretos em CLPs.

Outro trabalho que oferece uma contribuição significativa para a implementação de Sistemas a Eventos Discretos em Controladores Lógicos Programáveis é o estudo apresentado no trabalho (QUEIROZ; CURY, 2002). Esse trabalho explora a aplicação da Teoria de Controle Supervisório em uma célula de manufatura controlada por um CLP, por meio de uma metodologia que aproveita a modularidade da planta e dos modelos de especificações.

A metodologia proposta busca simplificar a aplicação do diagrama *ladder* no CLP, utilizando supervisores reduzidos implementados em uma estrutura de três níveis como visto na Figura 5. Essa estrutura permite a execução concorrente dos supervisores modulares e a interface entre o modelo teórico e o sistema real. A abordagem visa coordenar múltiplos subsistemas concorrentes na célula de manufatura, de forma que o sistema global atenda a uma série de especificações individuais e conjuntas.

A prova de conceito para a metodologia trabalhada envolve a célula de manufatura, composta por múltiplos subsistemas concorrentes, e tem como objetivo do controle supervisório a coordenação de tais subsistemas para que o sistema global obedeça a uma série de especificações individuais e conjuntas.

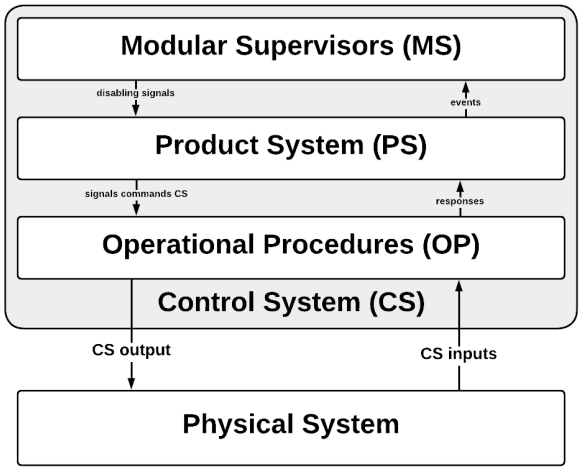

Figura 5 – Arquitetura de Controle Supervisório (QUEIROZ; CURY, 2002)

Assim, usando dessa abordagem modular local, a usina é representada como um conjunto de AFDs assíncronos, e as especificações são expressas localmente nas subplantas afetadas. Os supervisores ótimos são calculados com base nessas especificações locais e, em seguida, são reduzidos por meio de um algoritmo de minimização. O processo de síntese descrito anteriormente resulta em um conjunto de supervisores modulares locais reduzidos, representados por máquinas de estados finitos.

Com o objetivo de executar os supervisores modulares, o sistema de produto e os procedimentos operacionais de uma maneira que evite a composição de máquinas de estado, propõe-se que o sistema de controle seja programado em uma hierarquia de três níveis, conforme mencionado anteriormente. Essa estrutura pode ser implementada em linguagens de programação de CLP (*Ladder Diagram, Grafcet, etc.*), linguagens comuns (C++*, Java, etc.*) ou até mesmo diretamente em hardware.

Outro trabalho relevante que utiliza como base a arquitetura apresentada na Figura 5 e aborda a coordenação de diversos equipamentos em um sistema flexível de manufatura utilizando um CLP é descrito no artigo (VIEIRA et al., 2017). O método proposto nesse artigo utiliza uma arquitetura baseada em modelos de redes de Petri e máquinas de estado, que são empregados para modelar o comportamento dos subsistemas e do sistema como um todo.

Esse estudo detalha a implementação de um sistema de nível mais baixo, envolvendo a aplicação de teoria de controle supervisório em um ambiente complexo. Ele aborda dois problemas principais enfrentados nesse tipo de sistema. O primeiro problema consiste em controlar cada subsistema individualmente, levando em consideração seus próprios sensores, atuadores e controladores especializados, para executar uma sequência específica de atividades.

O segundo problema é a coordenação da operação simultânea desses subsistemas, visando produzir o que foi requisitado da maneira mais eficiente possível, ao mesmo tempo em que se garante a integridade e a segurança do sistema como um todo. Esse desafio requer uma

abordagem cuidadosa e estratégica, considerando a interação entre os subsistemas e a sincronização de suas atividades.

Para enfrentar esses desafios, o trabalho propõe a utilização de um SRA (Sistema de Representação de Processo) do sistema controlado, um conjunto de supervisores e um conjunto de POUs (*Program Organization Units*). O artigo apresenta uma série de algoritmos que fazem parte dos procedimentos sistemáticos para obter as variáveis e os *Function Blocks* (FB) que devem ser realizados para cada subsistema. Esses resultados são de grande relevância para o projeto em questão, pois fornecem uma base sólida e procedimentos bem definidos para a implementação da coordenação e controle dos subsistemas em um ambiente de manufatura flexível utilizando um CLP.

3.2 Field-Programmable Gate Arrays

Já na implementação com base em FPGAs, destaca-se o artigo (RAHMAN; ABU-GHAZALEH; NAJJAR, 2019), o qual apresenta experiências iniciais de implementação de um acelerador geral para Simulação de Eventos Discretos Paralelos (PDES) em um FPGA. O acelerador consiste em vários processadores capazes de processar eventos em paralelo, mantendo as dependências entre eles. Os eventos são automaticamente classificados por uma fila de eventos de classificação automática. O acelerador suporta simulação otimista, mantendo automaticamente o histórico de eventos e suportando reversões.

A arquitetura proposta possui limitações de escalabilidade local devido à comunicação e largura de banda da porta das diferentes estruturas. No entanto, foi projetada para permitir a conexão de vários aceleradores, a fim de ampliar a capacidade de simulação. O objetivo desse projeto é fornecer um acelerador PDES geral, em vez de um acelerador específico para um modelo ou classe de modelos. Para atingir essa generalidade, o PDES-A adota uma estrutura modular, onde vários componentes podem ser ajustados independentemente para alcançar o controle de fluxo de caminho de dados mais eficiente em diferentes modelos PDES.

Uma característica importante do PDES é o seu modelo de execução orientado a eventos, o qual não faz suposições sobre o tempo de execução dos eventos, uma vez que o tempo de processamento dos eventos pode variar entre diferentes modelos. Essa abordagem flexível permite adaptar o acelerador para diferentes cenários de simulação e maximizar o desempenho do sistema.

3.3 Microcontroladores

Na realidade dos microcontroladores, existem trabalhos que se dedicam a garantir uma boa implementação de SEDs. Uma metodologia para esse tipo de aplicação deve levar em consideração que a memória desses dispositivos é limitada, o que incentiva o uso de supervisores

modulares locais e estratégias para representação compacta dos modelos em memória, como foi feito no (LOPES et al., 2012).

Ao lidar com os sinais e eventos, o efeito avalanche é desconsiderado na implementação do microcontrolador, pois existe uma separação clara entre a estrutura de dados que representa o autômato e a implementação lógica do programa. No entanto, a implementação do microcontrolador deve levar em conta a ordem em que os eventos não controláveis ocorrem, a fim de evitar problemas de simultaneidade. Para lidar com questões relacionadas a causalidade, a implementação requer algum mecanismo que gerencie todos os eventos, enquanto a planta gera apenas os eventos não controláveis.

3.4 Conclusão

Com base nos estudos e projetos apresentados neste capítulo, é possível identificar algumas necessidades essenciais para a implementação física de Sistemas a Eventos Discretos, com o intuito de garantir um comportamento eficaz do sistema.

Uma dessas necessidades está relacionada ao tratamento dos sinais e eventos, abordando questões como simultaneidade e efeito avalanche. Na solução proposta neste trabalho, os eventos são interpretados por meio de sinais, e foi adotada a abordagem de avaliar apenas um evento por ciclo para evitar conflitos e interações indesejadas. Essa abordagem sequencial garante que os eventos sejam tratados de forma adequada e evita problemas decorrentes de simultaneidade.

Outro aspecto relevante é a consideração da causalidade dos eventos. Nesse sentido, os eventos controláveis são recebidos por um sistema externo, enquanto os eventos não controláveis são tratados pela planta. Essa distinção estabelece uma ordem de precedência na avaliação dos eventos, priorizando os eventos não controláveis. Essa estratégia assegura a consistência e a correta sequência de ações do sistema, contribuindo para o funcionamento adequado do controle automatizado.

Ao considerar essas necessidades no projeto e na implementação de sistemas físicos de Sistemas a Eventos Discretos, é possível alcançar um comportamento adequado e eficaz do sistema. O tratamento adequado dos sinais e eventos, evitando simultaneidade e efeito avalanche, juntamente com uma estratégia clara para a causalidade dos eventos, é fundamental para o correto funcionamento do sistema de controle automatizado.

É importante ressaltar que as soluções discutidas neste capítulo são aplicáveis ao contexto do Arduino, levando em consideração suas particularidades e restrições. A evolução da área demanda atualização constante e busca pelas melhores soluções para o desenvolvimento de projetos nesse campo. Portanto, a combinação dos conhecimentos teóricos apresentados no capítulo anterior com as abordagens já implementadas na literatura proporciona uma base sólida para a implementação do sistema proposto para o Arduino.

4 Metodologia

Nos capítulos anteriores, foram abordadas as bases e os conhecimentos relevantes para o desenvolvimento da solução que será implementada neste projeto de final de curso. Agora, neste capítulo, será fornecido um detalhamento da arquitetura escolhida para a solução, destacando os procedimentos selecionados para o desenvolvimento e as principais decisões de projeto.

Inicialmente, será abordada a estrutura do projeto, descrevendo a organização e as decisões relacionadas à implementação de baixo nível. Será explicado como as estruturas principais de um SED foram modeladas para o Arduino, levando em consideração as características específicas dessa plataforma. Além disso, serão apresentadas as decisões de implementação e a arquitetura escolhida para a conversão e para o fluxo principal implementado no Arduino.

Ao final deste capítulo, espera-se que o leitor tenha uma compreensão clara e aprofundada do processo de desenvolvimento e implementação do sistema de controle supervisório no Arduino. As informações apresentadas serão essenciais para avaliar os resultados obtidos e realizar uma análise crítica do sistema, contribuindo para a conclusão geral do projeto.

4.1 Organização do projeto

Esta seção tem como objetivo abordar a organização do projeto, levando em consideração alguns fatores principais que foram considerados antes do desenvolvimento.

A primeira discussão aborda a estruturação do projeto, descrevendo a quantidade de arquivos desenvolvidos, seus nomes e suas funções dentro da solução. Será fornecida uma visão geral dos arquivos que compõem o projeto, destacando a importância de cada um e como estão interligados para garantir o funcionamento adequado da solução.

Serão apresentadas também como o projeto lidou com as implementações de baixo nível, considerando que um SED modela o comportamento em alto nível de um sistema. Será ressaltada a necessidade de reservar parte do programa resultante para a implementação desse comportamento pelo usuário final. Isso permite que a solução seja adaptada de acordo com as características específicas da planta, sem a necessidade de modificar todos os arquivos criados.

Essas discussões visam fornecer uma compreensão abrangente da organização do projeto, incluindo a estrutura, as partes fixas e as partes personalizáveis da solução. Essas informações serão essenciais para avaliar a eficiência e a flexibilidade do sistema desenvolvido.

4.1.1 Estrutura do Projeto

Ao desenvolver o presente projeto foram considerados diversos pontos para garantir a eficiência, flexibilidade e facilidade de manutenção da solução. Inicialmente foi pensando em armazenar todos esses códigos como constantes no programa principal, contudo isso resultaria em um código extenso e difícil de gerenciar e ao utilizar um arquivo externo, é possível organizar e estruturar a solução mantendo o código-fonte mais limpo e legível.

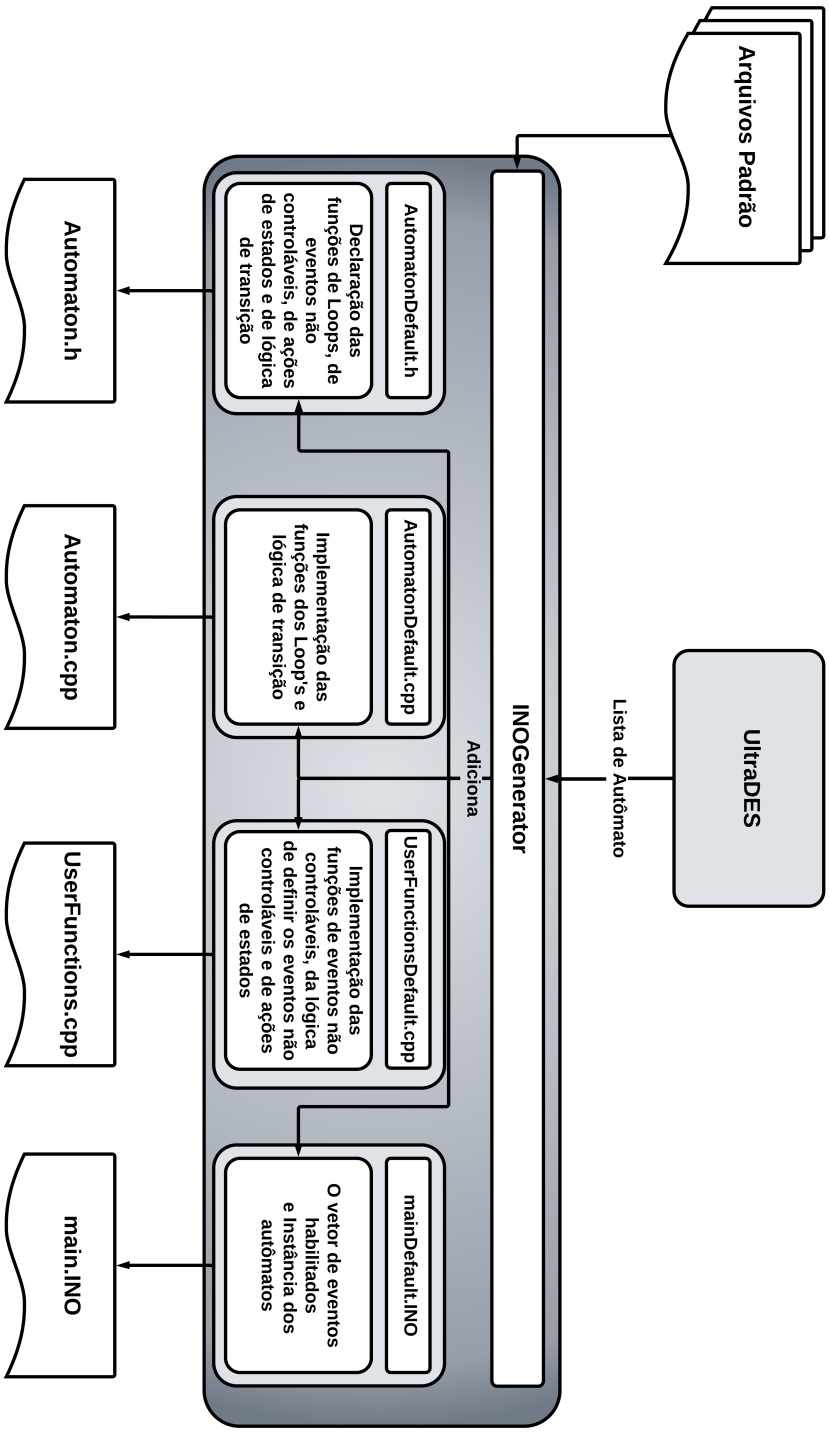

Figura 6 – Estrutura dos arquivos da solução

Dessa forma optou-se por armazenar os códigos base necessários para a conversão, que seriam comuns a todas as implementações de uma lista de SEDs em sistema embarcados, em cinco a arquivo separados. Isso proporciona flexibilidade e facilita a manutenção, pois os dados podem ser modificados sem a necessidade de alterar o código principal da solução, além disso que a separação entre os códigos torna mais claro o processo de conversão e facilita a compreensão do programa como um todo. Assim a solução fica como a estrutura presente na figura 6 proporciona clareza, modularidade e facilidade na manutenção e evolução do projeto.

4.1.1.1 `INOGenerator.cs`

Dentro do projeto, o arquivo `INOGenerator.cs` desempenha um papel fundamental na implementação principal da solução. Ele é escrito em C#, uma escolha que ocorre devido ao fato de o UltraDES ter sido desenvolvido nessa linguagem e por esse motivo é mais simples de manipular os dados proveniente do mesmo. O objetivo desse arquivo é traduzir os autômatos do UltraDES, convertendo sua lógica para uma forma compatível para o Arduino. Ele receberá os demais arquivos e adicionará a esses a logica da lista de autômatos provenientes do UltraDES.

Segundo a Figura 6, pode-se observar o funcionamento dessa função no projeto, cujo objetivo é realizar a transformação dos arquivos padrões que funcionam como *templates*, adaptando e integrando a lógica dos autômatos a serem convertidos, sendo assim:

- `mainDefault.INO` torna-se `main.INO`
- `AutomatonDefault.h` torna-se `Automaton.h`
- `AutomatonDefault.cpp` torna-se `Automaton.cpp`
- `UserFunctionsDefault.cpp` torna-se `UserFunctions.cpp`

4.1.1.2 `mainDefault.INO`

Esse arquivo *template* desempenha um papel fundamental ao coordenar a sequência de execução do programa final. Assim o arquivo `mainDefault.INO` da solução é responsável por iniciar as a logica necessárias e chamar as demais funções de acordo com a lógica do conjunto de autômatos traduzidos. Dessa forma, após a conversão, o `main.INO` garantirá o correto fluxo de execução e interação entre os diferentes componentes do sistema.

É importante ressaltar que o arquivo `main.INO` passa por poucas alterações durante a execução do `INOGenerator.cs`, uma vez que sua implementação abrange a lógica geral para qualquer SED, sendo somente adicionado o vetor de eventos habilitados para cada estado de cada automato e a instância dos objetos autômatos. Isso significa que sua estrutura é projetada para ser reutilizável, adaptando-se aos diferentes autômatos e suas respectivas necessidades.

4.1.1.3 `AutomatonDefault.h`

O arquivo `AutomatonDefault.h` é um *header*, ou seja, um arquivo de cabeçalho, em que são declaradas as estruturas, classes e funções que serão definidas e implementadas em outros arquivos. Por meio dele é possível organizar e centralizar as declarações do projeto facilitando a leitura e manutenção do código, pois as declarações estão agrupadas em um único local, tornando mais fácil a visualização e compreensão do que está sendo utilizado no projeto.

Como *header* todas as funções contidas na solução estão declaradas nesse arquivo, é importante observar que a classe "*Automaton*" e suas funções membros fornecem uma estrutura consistente e reutilizável que permanece inalterada em todas as execuções do programa final. Por outro lado, as funções de transição de estado e de ação são adicionadas dinamicamente, dependendo da configuração de cada autômato permitindo que cada autômato tenha sua própria lógica de transição de estado e de ação, adaptando-se às necessidades específicas do sistema em que está sendo utilizado.

4.1.1.4 `AutomatonDefault.cpp`

Já o `AutomatonDefault.cpp` contém a implementação das funções dos *loops* de cada autômato, bem como a lógica de transição dos estados, sendo que cada automato possui a sua própria função de *Loop* e de transição.

Esse arquivo foi projetado com a intenção de não ser necessário que o usuário final faça alterações diretas nele, como a intenção de garantir a estabilidade e a consistência do sistema. Ele serve como uma base sólida que executa a lógica padrão dos autômatos, permitindo que o usuário se concentre na personalização de outros aspectos do sistema.

4.1.1.5 `UserFunctionsDefault.cpp`

Por fim o arquivo `UserFunctionsDefault.cpp` contém a implementação das funções que interpretam os eventos não controláveis inserindo os mesmo em um vetor de eventos, a lógica de definição desses eventos e as ações dos estados. Essas implementações estão disponíveis para alteração pelo usuário final e são responsáveis pela implementação de baixo nível do sistema.

A escolha de permitir que o usuário final altere essas implementações tem como objetivo fornecer flexibilidade e adaptabilidade ao sistema. Dessa forma, o usuário pode personalizar as ações dos estados e a lógica de eventos não controláveis de acordo com as necessidades específicas do projeto ou da planta em que o sistema está sendo implementado.

4.1.2 Implementação de Baixo Nível

A implementação de baixo nível da solução desenvolvida foi designada como responsabilidade do usuário final, devido a dois principais motivos: Em primeiro lugar, a solução

idealizada visa ser compatível com uma ampla gama de microcontroladores, que possuem periféricos distintos e diferentes números de pinos de entrada e saída. É inviável ter uma implementação de baixo nível que funcione de maneira idêntica para todos os dispositivos. Além disso, existe uma diversidade de sistemas que podem ser modelados pelos mesmos autômatos, tornando impossível determinar qual dos vários sistemas está sendo implementado.

Consequentemente, o projeto destinou uma parte do programa resultante para que o usuário final pudesse implementar esse comportamento específico, adaptando-o às características particulares da planta, sem a necessidade de modificar todos os arquivos criados. Essa abordagem permite que o usuário personalize a implementação de acordo com as necessidades específicas do hardware em uso e das características do sistema que está sendo controlado. Dessa forma, o projeto oferece flexibilidade e facilidade de adaptação, permitindo que o usuário final ajuste a solução de acordo com suas próprias exigências.

Ao reservar essa parte da implementação para o usuário final, é possível abranger uma variedade maior de dispositivos e sistemas, garantindo a compatibilidade e a aplicabilidade da solução em diferentes contextos. Essa abordagem também promove a reutilização dos componentes principais do programa, evitando a necessidade de modificar todos os arquivos criados, o que facilita a manutenção e a escalabilidade do projeto.

Portanto, ao conceder ao usuário final a responsabilidade pela implementação de baixo nível, o projeto busca oferecer uma solução flexível, adaptável e compatível com uma ampla gama de dispositivos e sistemas, ao mesmo tempo em que simplifica o processo de personalização e manutenção do sistema.

Para garantir a modularidade da solução, todas as áreas em que o usuário pode interagir foram agrupadas em um único arquivo, o `UserFunctionsDefault.cpp`. Essa abordagem evita a necessidade de alterar vários arquivos e reduz a possibilidade de perder o comportamento de alto nível garantido pela solução.

O arquivo `UserFunctionsDefault.cpp` contém as seguintes funcionalidades disponiveis para edições:

- Funções que determinam as ações dos estados: Por padrão, essas funções realizam apenas o envio de uma mensagem por comunicação serial, mas podem ser alteradas de acordo com as necessidades da planta em questão.
- Função responsável por configurar os pinos do Arduino: Essa função permite que o usuário defina a configuração dos pinos do Arduino de acordo com a sua planta específica, adaptando-se à interface de hardware necessária.
- Função que configura o gerenciamento de ocorrência de eventos, sejam eles controláveis ou não. Essa função permite que o usuário defina como os eventos são gerenciados adaptando-se às características dos eventos específicos da planta.

Essa abordagem modular e personalizável do código permite que o usuário final adapte a solução de acordo com as particularidades de cada planta real que está sendo modelada. Ao fornecer essa flexibilidade, o projeto se torna mais robusto e capaz de lidar com uma variedade maior de cenários, ao mesmo tempo em que mantém a consistência do sistema como um todo.

4.2 Tradução das Estruturas de SEDs

Para o desenvolvimento da aplicação que traduz a lógica de um SED para o Arduino, é crucial encontrar uma forma eficiente de representar os estados, eventos, transições e autômatos. Essa representação adequada é essencial para garantir a execução correta do sistema no hardware do Arduino, levando em consideração suas limitações de memória e processamento.

Inicialmente, a abordagem considerada foi a utilização do paradigma de orientação a objetos, similar ao empregado pelo UltraDES, para representar as estruturas no C++. No entanto, levando em conta as restrições de memória em sistemas embarcados, decidiu-se adotar uma abordagem alternativa visando otimizar o uso de memória e processamento no Arduino. Isso não significa que a orientação a objetos tenha sido completamente abandonada, mas sim reduzida para alcançar um melhor desempenho.

Uma das decisões tomadas foi utilizar apenas uma classe para representar os autômatos, simplificando a estrutura de dados e reduzindo a sobrecarga de memória associada à orientação a objetos. Para as demais estruturas, como estados, eventos e transições, foram adotadas estratégias diferentes para otimização.

Essa abordagem alternativa busca encontrar um equilíbrio entre a representação eficiente dos elementos do SED e a capacidade limitada do hardware do Arduino. Dessa forma, é possível desenvolver uma aplicação que execute corretamente a lógica do sistema, levando em consideração as restrições de recursos do Arduino.

4.2.1 Estados

Em um SED, um estado representa uma configuração específica na qual o sistema se encontra em determinado momento, contendo informações relevantes para o comportamento e evolução do sistema. Para traduzir esses estados para o contexto do Arduino, duas características principais foram escolhidas: um identificador único e uma ação associada a cada estado.

Para representar o identificador de cada estado, foi utilizado de um número inteiro de 16 bits. Essa escolha permite um limite superior de 65535 estados possíveis, o que abrange a maioria das aplicações, considerando as possíveis complexidades do sistema em questão. Mesmo que exista diversas aplicações com mais de $2^{16} - 1$ estados, essa escolha garante uma capacidade adequada para representar um número suficientemente grande de estados, sendo inclusive maior do que a capacidade do Arduino.

Além disso, foi implementada uma função para representar a ação associada a cada estado. Essas funções serão responsáveis por executar tarefas específicas relacionadas a cada estado. Cada função pode ser definida de acordo com as necessidades da aplicação, permitindo a execução de ações específicas em resposta a transições de estado.

Essa abordagem de representação dos estados no Arduino, com identificadores únicos e funções associadas, permite a correta execução da lógica do sistema e o gerenciamento das tarefas específicas de cada estado, da maneira mais simples e ocupando o menor espaço na memoria do dispositivo. Além de facilitar a programação e a compreensão do código, uma vez que cada estado é representado de forma clara e concisa tornando o código mais modular e facilitando a manutenção e atualização do sistema.

4.2.2 Eventos

A abordagem adotada para representar os eventos no Arduino consiste na escolha de um número inteiro de 32 *bits*. Cada evento é representado pela posição do número inteiro, onde um *bit* 1 indica que o evento ocorreu e um bit 0 indica que o evento não ocorreu. Essa representação utiliza uma estratégia de máscara de bits para realizar operações envolvendo eventos, proporcionando uma solução eficiente para lidar com a ocorrência de eventos em um Sistema a Eventos Discretos.

A utilização de bits para representar os eventos traz diversas vantagens. Primeiramente, essa abordagem simplifica a verificação dos eventos no sistema. Por meio de operações lógicas com as máscaras de bits apropriadas, é possível identificar facilmente quais eventos estão ocorrendo. Através da aplicação de operações como AND, OR e NOT, é possível combinar, comparar e manipular os eventos de forma eficiente.

Os eventos controláveis são obtidos através de um sinal externo, por padrão se espera um valor da porta serial, onde um valor inteiro é recebido e representa os eventos controláveis que se deseja que ocorram. Através da máscara de bits, é possível verificar quais eventos foram acionados e atualizar o estado correspondente de acordo.

Já os eventos não controláveis são adquiridos por meio de uma função especial que verifica se determinado fenômeno da planta ocorreu ou não. Essa função é responsável por obter os eventos não controláveis e atualizar o estado correspondente, permitindo que o sistema reaja de acordo com esses eventos.

Essa estrategia de representação dos eventos, utilizando máscara de bits, é uma forma eficiente e compacta de lidar com a ocorrência de eventos em um SED, permitindo a correta execução da lógica do sistema e o gerenciamento dos eventos de forma adequada.

4.2.3 Transição

Uma transição em um SED descreve a mudança do sistema do estado atual para outro, refletindo as consequências da ocorrência do evento. Dessa forma as transições foram implementadas por meio de uma função que recebe o estado atual e o último evento que ocorreu, a função verifica o estado atual e determina qual será o próximo estado de acordo com as transições definidas no autômato.

Essa abordagem é especialmente útil em sistemas embarcados, como o Arduino, onde a eficiência de processamento e o uso de recursos de memória são importantes. A implementação das transições por meio de uma função é uma forma simples e otimizada de lidar com as mudanças de estado no sistema, garantindo a correta execução da lógica do autômato.

4.2.4 Autômatos

A classe "*Automaton*" foi desenvolvida para representar um autômato finito genérico no contexto da tradução para o Arduino. Essa classe oferece flexibilidade ao permitir a configuração do autômato com um número variável de estados, eventos habilitados e funções de transição e execução. Para facilitar a compreensão da estrutura da classe, foi criado um diagrama de classe, conforme apresentado na Figura 7.

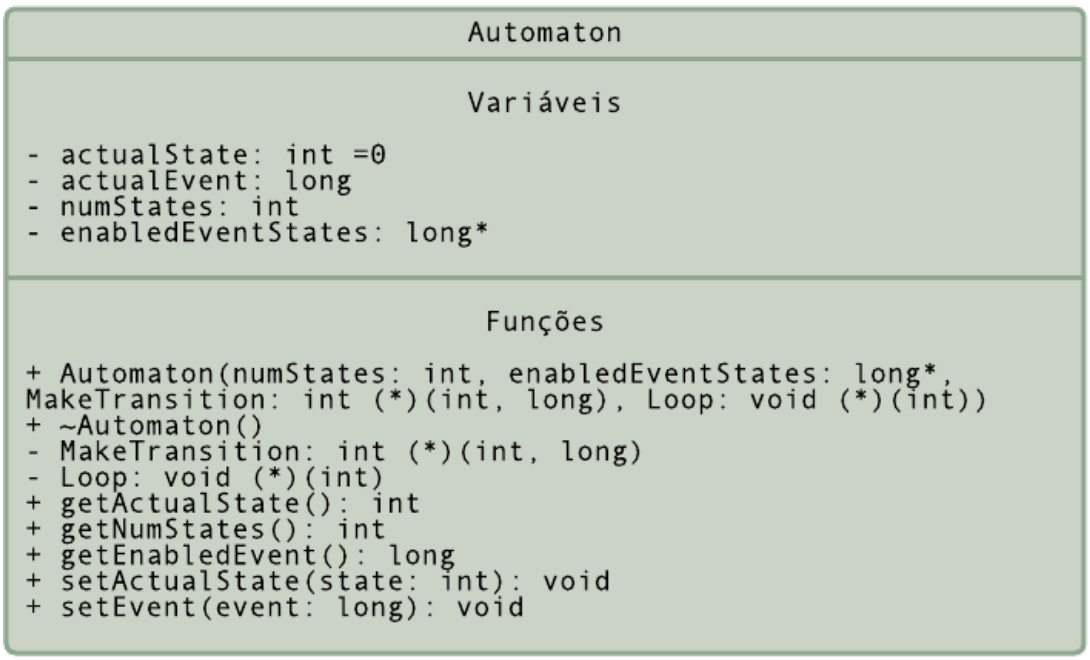

Figura 7 – Diagrama de Classe de "*Automaton*"

No diagrama, é possível observar que a classe possui quatro atributos privados: um valor inteiro de 16 bits para representar o estado atual do autômato (inicializado como zero),

um valor de 32 bits para armazenar o último evento ocorrido, o número total de estados e uma lista de eventos habilitados por estado. A classe possui as seguintes funções:

- *Automaton()*: O construtor da classe, que recebe como parâmetros o número de estados do autômato, um ponteiro para a lista de valores de eventos habilitados, um ponteiro para a função de transição de estados e um ponteiro para a função de execução do estado atual. Esses parâmetros são utilizados para configurar o autômato.
- *∼Automaton()*: O destrutor do objeto, responsável por liberar o espaço alocado na memória.
- *getActualState()*: Função que retorna o estado atual do autômato.
- *getNumStates()*: Função que retorna o número total de estados do autômato.
- *getEnabledEvent()*: Função que retorna os eventos habilitados no estado atual do autômato.
- *setActualState(int state)*: Função que define o estado atual do autômato com base no valor fornecido como argumento.
- *(*MakeTransition)(int state, long event)*: Ponteiro para a função de transição de estados. Esse ponteiro é inicializado no construtor, permitindo que o autômato execute as transições corretamente.
- *(*Loop)(int state)*: Ponteiro para a função de loop, responsável por chamar as ações de estado com base no estado atual. Esse ponteiro é atribuído no construtor.

Essas funcionalidades permitem a correta execução da lógica do autômato, gerenciando os estados, os eventos e as transições de acordo com as configurações estabelecidas.

4.3 Arquitetura

Nesta seção do capítulo de metodologia, serão apresentados os fluxogramas que representam a arquitetura das partes fundamentais do projeto. Esses fluxogramas visuais descrevem o fluxo de informações e as etapas envolvidas no processamento dos dados, proporcionando uma visão clara da estrutura e das interações dos componentes do sistema. Dois fluxogramas serão apresentados: o fluxograma da lógica de conversão e o fluxograma da rotina principal.

Esses fluxogramas são ferramentas importantes para compreender o funcionamento do sistema e através deles é possível visualizar de forma clara e organizada o funcionamento das partes fundamentais do projeto, auxiliando no desenvolvimento e na compreensão do sistema como um todo.

4.3.1 Arquitetura da Função de Tradução

O fluxograma da lógica de conversão representa o processo de conversão da lógica do SED para a implementação no Arduino. Ele descreve as etapas envolvidas na tradução dos estados, eventos, transições e autômatos, levando em consideração as restrições de memória e processamento do Arduino. Esse fluxograma permite compreender como as estruturas do SED são mapeadas para as estruturas utilizadas na implementação no Arduino, garantindo a correta execução da lógica do sistema.

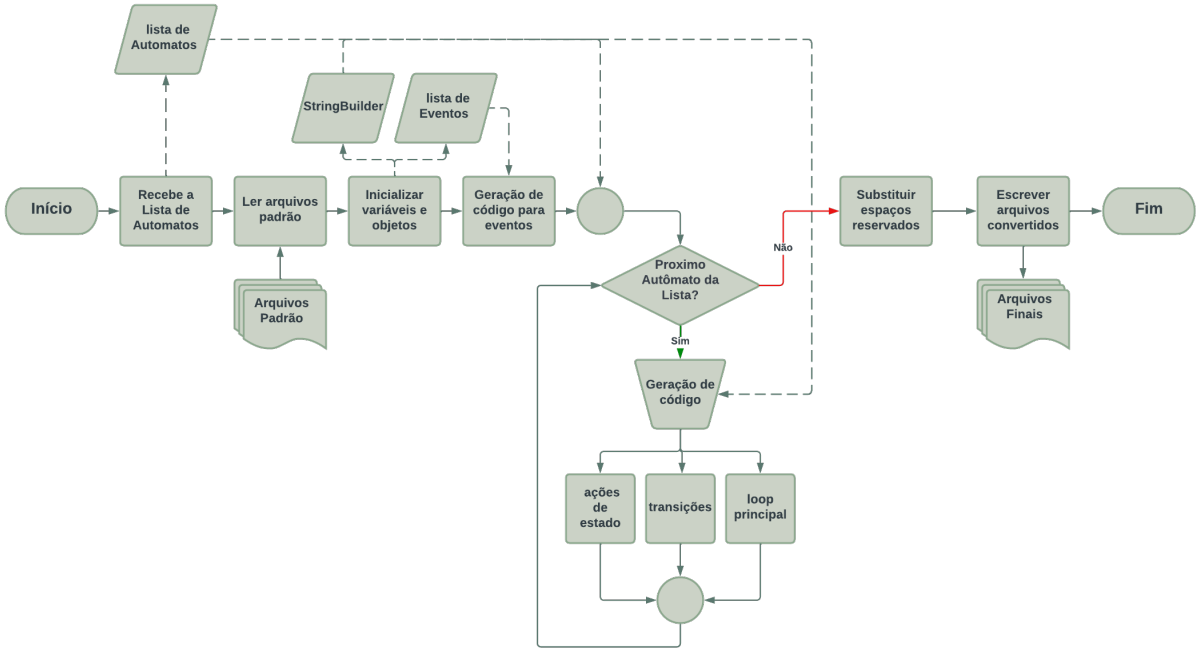

Figura 8 – Fluxograma de conversão dos autômatos finitos para código em Arduíno

As principais etapas dessa arquitetura são:

1. **Leitura dos arquivos padrão:** Nessa etapa, são lidos os arquivos de *template* padrão que contêm o código base no formato *INO*, bem como as estruturas e funções pré-definidas.

2. **Inicialização de variáveis e estruturas:** Nessa etapa, são inicializadas as variáveis e estruturas que serão utilizadas para armazenar informações sobre os autômatos e eventos.

3. **Geração de códigos para eventos:** Nessa etapa, os eventos dos autômatos são traduzidos em sequências binárias. Cada evento é mapeado para uma sequência de bits única, permitindo uma representação compacta dos eventos. Em seguida, são criadas funções para lidar com os eventos não controláveis. Para cada evento não controlável, são geradas funções que retornam "true"ou "false"dependendo da ocorrência do evento.

4. **Geração do código para todos os autômatos da lista:** Para cada autômato na lista são obtidas as informações relevantes, como o número de estados, a tabela de eventos habilitados, a função de transição de estados e a função de *loop*. Com base nessas informações,

são criadas as seções de código correspondentes ao autômato. Primeiramente, declara-se a variável do autômato, usando o tipo de dados adequado para armazenar o estado atual, e inicializa-se a variável com o estado inicial do autômato.

5. **Substituição de texto nos arquivos de saída:** Nessa etapa, ocorre a substituição de marcadores de posição nos arquivos de template com o código gerado para cada parte do programa. Isso inclui a substituição das funções de eventos, transições de estado, loops e outras informações específicas dos autômatos.

6. **Escrita dos arquivos de saída:** Nessa etapa, os arquivos de saída são gravados no disco. Os arquivos finais contêm o código completo do programa no formato INO, pronto para ser carregado em um dispositivo Arduino.

4.3.2 Arquitetura da Rotina Principal

O fluxograma da rotina principal descreve a sequência de ações executadas pelo sistema durante o seu funcionamento normal. Ele engloba as principais funcionalidades do sistema, como a leitura de eventos, a verificação de transições e a execução das ações associadas a cada estado. Esse fluxograma oferece uma visão detalhada do fluxo de controle do sistema, destacando os pontos de interesse e os principais passos envolvidos no processamento dos eventos e transições.

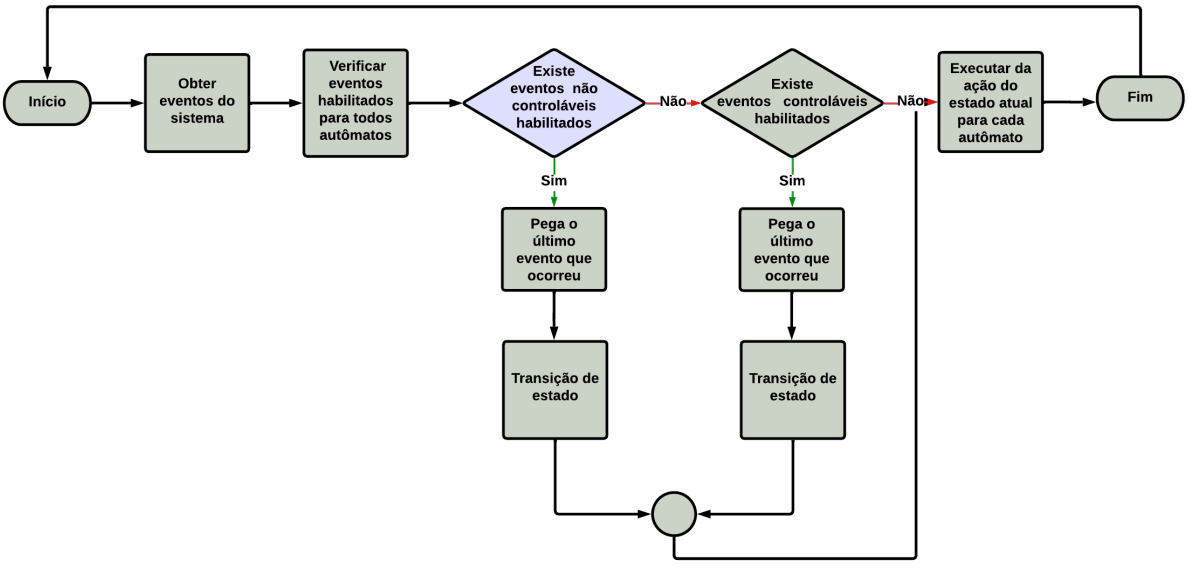

Figura 9 – Fluxograma da rotina principal

O ciclo principal da rotina principal é repetido continuamente, permitindo que os autômatos processem eventos, realizem transições de estado e executem suas lógicas específicas. Isso assegura o correto funcionamento do sistema baseado em autômatos. As principais etapas dessa arquitetura são:

1. **Obtenção dos eventos do sistema:** Nessa etapa, são obtidos os eventos do sistema, divididos em três conjuntos: *eventControllable* que representa os eventos controláveis, *eventUncontrollable* que representa os eventos não controláveis e *eventEnabled* que é formado pelos eventos habilitados no estado atual do autômato.

2. **Verificação de eventos habilitados:** Nessa etapa, é realizada uma verificação para determinar quais eventos estão habilitados. Para cada autômato existente no sistema, um *loop* é percorrido. Dentro desse *loop*, é realizada uma operação de *bit* a *bit* entre o conjunto de eventos habilitados (*eventEnabled*) e o conjunto de eventos controláveis (*eventControllable*) ou o conjunto de eventos não controláveis (*eventUncontrollable*), dependendo do caso. Essa operação tem como objetivo identificar quais eventos estão habilitados para o sistema como um todo.

3. **Execução de transição de estados para eventos não controláveis:** Se houver eventos habilitados não controláveis (*eventEnabledUncontrollable*), é realizado um *loop* para verificar qual é o primeiro evento habilitado dessa categoria e tendo em mãos esse evento, é executada a transição de estado em cada autômato existente no sistema. Essa transição de estado é realizada chamando a função *MakeTransition()* de cada autômato, passando como parâmetros o estado atual e o evento habilitado.

4. **Execução de transição de estados para eventos controláveis:** Caso não existam eventos habilitados não controláveis, é verificado se existem eventos habilitados controláveis (*eventEnabledControlable*). Novamente, é feito um *loop* para identificar qual é o primeiro evento habilitado controlável. Essa transição de estado é realizada chamando a função *MakeTransition()* de cada autômato, passando como parâmetros o estado atual e o evento habilitado.

5. **Execução do *loop* do autômato:** Por fim, é realizado um *loop* para executar a função *Loop()* de cada autômato. Essa função recebe o estado atual e executa a ação referente ao mesmo.

5 Desenvolvimento e Resultados

No presente capítulo, será apresentada a implementação prática da metodologia descrita no capítulo anterior. O objetivo é detalhar os passos seguidos para traduzir os fluxogramas em código e como cada etapa do processo foi implementada, levando em consideração a estrutura do projeto proposta e a implementação de baixo nível.

Nesse capítulo, será abordada em detalhes a função "*ConvertDEStoINO*" do arquivo "*INOGenerator*". Será mostrado o código correspondente a cada etapa do fluxograma da lógica de conversão presente na Figura 8. Será destacada a lógica por trás de cada trecho de código, explicando as decisões tomadas e as principais estruturas utilizadas.

Também serão apresentado os *templates* desenvolvidos explicando a lógica por trás dos mesmo, com o intuito de demonstrar os locais onde os arquivos padrões serão editados pela função principal. De forma semelhante será trabalhado o desenvolvimento direcionado da implementação da rotina principal no Arduino, de acordo com o fluxograma apresentado. Será demonstrado o código desenvolvido para executar as ações e funcionalidades previstas na lógica da rotina principal.

Essa análise prática será fundamental para avaliar a eficácia e a viabilidade do sistema, além de fornecer *insights* valiosos para futuras melhorias e aprimoramentos do projeto.

5.1 Implementação da função *ConvertDEStoINO*

Nesta seção, será apresentado a implementação da função *ConvertDEStoINO* que desempenha o papel principal do presente projeto em desenvolvimento. Essa função é responsável por converter autômatos finitos determinísticos em código*INO*,que é utilizado em placas Arduino.

O objetivo principal é explorar em detalhes o processo de implementação da função *ConvertDEStoINO*, de forma a compreender sua estrutura e funcionamento. Dessa forma tem-se :

1. **Leitura dos arquivos padrão:**

   Nessa etapa pode-se observar que o código apresentado realiza a leitura dos quatro arquivos padrão, que servirão como base para a geração de arquivos resultantes. Essa abordagem permite a personalização do código gerado, pois o conteúdo dos arquivos padrão é utilizado como modelo. Essa estratégia facilita a manutenção e reutilização do código, pois as alterações podem ser feitas nos arquivos padrão, evitando a necessidade de modificar diretamente o código fonte gerado.

2. **Inicialização de variáveis e estruturas:**

   Nesse passo diversas variáveis do tipo *StringBuilder* são inicializadas para construir diferentes partes do código resultante. Cada variável tem um propósito específico na lógica de implementação, como a construção de ações de estado, lógica de transição, *loops*, vetores e funções dos eventos não controláveis. O uso de *StringBuilder* permite uma concatenação eficiente de *strings*, evitando a criação excessiva de objetos *string* intermediários. Essa abordagem é útil para construir o código resultante de forma mais eficiente em termos de desempenho e utilização de memória.

3. **Geração de códigos para eventos:**

   - **Tradução dos Eventos:**

     Nessa parte do projeto uma lista de eventos abstratos *listOfEvents* é criada para armazenar os eventos presentes em todos os autômatos da lista fornecida. Em seguida, é criado um dicionário chamado *eventMap* para mapear cada evento abstrato a uma sequência de caracteres. O uso de um dicionário para armazenar o mapeamento permite um acesso rápido e direto às sequências de *bits* de cada evento.

   - **Geração dos códigos para eventos não controláveis:**

     Nessa etapa são criadas as funções que irão gerir o recebimento dos eventos não controláveis e são reservadas para a edição do usuário final. Ao passar pela lista de eventos é verificado se o evento não é controlável, se não for, o código cria uma *string* contendo o nome do evento e a adiciona o texto de código correspondente no *StringBuilder*.

```
início
    Criar dicionário eventMap para mapear eventos para strings;
    Obter número de eventos em listOfEvents;
    para cada evento em listOfEvents: faça
        Armazene sequência binária correspondente ao evento em eventMap.
    fim
    para cada evento em listOfEvents: faça
        se evento não for controlável: então
            Crie a etapa da getEventosNãoControláveis referente a esse evento;
        fim
    fim
fim
```

Algoritmo 1: `ConvertDEStoINO`: Geração dos códigos para eventos não controláveis

4. **Geração do código para todos os autômatos da lista:**

   Essa lógica implementa um *loop* que percorre a lista de autômatos. Para cada autômato na lista, é obtido o número de estados e armazenado em uma variável. Em seguida, são

construídas várias *strings* contendo declarações de funções e lógicas de transição para cada estado do autômato. É criado um dicionário para mapear os estados para valores inteiros, em seguida, são percorridos os estados do autômato, obtendo as transições e eventos correspondentes. Além disso, são construídas outras *strings* contendo vetores de eventos habilitados e a função de transição. Por fim, é criado um objeto para cada autômato, contendo as informações e funções necessárias.

```
início
    para cada automato na Lista de automato: faça
        Obter o número de estados do automato;
        para cada estado do automato: faça
            Cria a definição da ação referente ao estado;
            Cria a implementação da ação referente ao estado;
            Adiciona a um vetor os eventos habilitados desse estado;
        fim
        Cria o vetor de ações de estado;
        Produz a declaração da função de loop;
        Produz a implementação função de loop que invoca o vetor de ações de estado;
        Mapeia os estados para números inteiros;
        Produz a declaração da função de transição;
        Produz a implementação da função de transição usando os estados e eventos
          mapeados;
    fim
fim
```

Algoritmo 2: `ConvertDEStoINO`: Geração do código para todos os autômatos da lista

5. **Substituição de texto nos arquivos de saída:**

   O código substitui *placeholders* nos arquivos padrão pelos blocos de código gerados para cada autômato. Isso inclui substituir trechos de código marcados com comentários como "*// ADD-STATE-ACTION*", "*// ADD-TRANSITION-LOGIC*", "*// ADD-AUTOMATON-LOOP*", "*// ADD-EVENT-UNCONTROLLABLE*", entre outros. A substituição desses *placeholders* permite que o código gerado seja integrado aos arquivos existentes e funcionais.

6. **Escrita dos arquivos de saída**

   Essa é a etapa responsável por escrever os resultados da conversão nos arquivos correspondente.

5.2 Implementação da Rotina Principal

O objetivo desta seção é detalhar os passos seguidos para implementar a rotina principal no Arduino, de acordo com o fluxograma apresentado na Figura 9. Essa função controla o comportamento de autômatos de estados finitos.

A Rotina Principal começa obtendo os eventos do sistema e verifica quais eventos estão habilitados nos autômatos. Com base nisso, realiza transições de estado nos autômatos. Primeiro, verifica eventos não controláveis e executa as transições correspondentes. Se não houver eventos não controláveis, verifica eventos controláveis e executa as transições apropriadas. Por fim, chama a função *Loop()* de cada autômato, passando o estado atual como parâmetro.

1. **Obtenção dos eventos do sistema:**

```
// Get the system events
long eventControllablble = getEventControllablble();
long eventUncontrollable = getEventUncontrollable();
long eventEnabled = 0;
```

Algoritmo 3: `mainDefault.INO`: Obtenção dos eventos do sistema

2. **Verificação de eventos habilitados:**

```
// Check the enabled events and if any enabled event was detected
for (int i = 0; i < NUMBER_AUTOMATON; i++)
{eventEnabled |= automata[i].getEnabledEvent();}
long eventEnabledControlable =eventControllablble & eventEnabled;
long eventEnabledUncontrollable = eventUncontrollable & eventEnabled;
```

Algoritmo 4: `mainDefault.INO`: Verificação de eventos habilitados

3. **Execução de transição de estados para eventos não controláveis:**

```
if (eventEnabledUncontrollable > 0){
 for (int i = 0; i < NUMBER_EVENT; i++){
 // Get the First Uncontrollable Event Enabled
  long event = eventEnabledUncontrollable & (1L << i);
  if (event > 0){ // Execute the state transition for each automaton
   for (int j = 0; j < NUMBER_AUTOMATON; j++){
    int actualState = automata[i].getActualState();
    int nextState =
    automata[i].MakeTransition(actualState, eventEnabled);
    automata[i].setActualState(nextState);}
   break;}}}
```

Algoritmo 5: `mainDefault.INO`: Transição de estados para eventos não controláveis

4. **Execução de transição de estados para eventos controláveis:**

```
else
{
 if (eventEnabledControlable > 0)
 {
  for (int i = 0; i < NUMBER_EVENT; i++)
  { // Get the First Uncontrollable Event Enabled
   long event = eventEnabledControlable & (1L << i);
   if (event > 0)
   { // Execute the state transition for each automaton
    for (int j = 0; j < NUMBER_AUTOMATON; j++)
    {
     int actualState = automata[i].getActualState();
     int nextState =
     automata[i].MakeTransition(actualState, eventEnabled);
     automata[i].setActualState(nextState);
    }
    break;
}}}}
```

Algoritmo 6: `mainDefault.INO`: Transição de estados para eventos controláveis

5. **Execução do *loop* do autômato:**

```
// Execute the Loop function for each automaton
for (int i = 0; i < NUMBER_AUTOMATON; i++)
{
    int actualState = automata[i].getActualState();
    automata[i].Loop(actualState);
}
```

Algoritmo 7: `mainDefault.INO`: Execução do *loop* do autômato

Cada trecho de código representa a implementação da respectiva etapa no ciclo principal de execução do programa.

5.3 Implementação dos *templates*

A presente seção tem como objetivo apresentar a implementação dos arquivos padrões com base no seção 4.1.1. Será abordado cada um dos *templates* implementados, descrevendo as funcionalidades e características de cada um.

5.3.1 `AutomatonDefault.h`

O código observado é um cabeçalho (*header file*) em C++ que define a classe *Automaton*. Nessa seção serão explicados as principais partes do código relacioná-las ao conceito de Sistemas a Eventos Discretos.

1. **Declaração do tipo de Ação Genérica:** Essa declaração define um tipo de dado chamado *GenericAction*, que é um ponteiro para uma função que não retorna valor e não possui parâmetros.
2. **Classe Automaton:** Implementa a classe desenhada na seção 4.2.4.
3. **Funções auxiliares:**
    - **void setupPin():** Ela é usada para configurar os pinos específicos em uma placa Arduino.
    - **long getEventControllablble():** Essa função retorna o valor dos eventos controláveis do automato.
    - **long getEventUncontrollable():** Essa função retorna o valor dos eventos não controláveis do automato.

No código, existem também comentários que indicam a intenção de adicionar outras funcionalidades relacionadas a ações de estado, eventos incontroláveis, lógica de transição de estado, entre outros. No entanto, as implementações dessas funcionalidades não estão presentes no código.

5.3.2 `AutomatonDefault.cpp`

Aqui encontra-se a implementação dos métodos da classe *Automaton* que foram declarados no arquivo de cabeçalho "`AutomatonDefault.h`".

1. **Construtor e destrutor:**
    - O construtor é aqui implementado inicializando os membros da classe com os parâmetros fornecidos. Isso permite definir o número de estados do automato (*numStates*), a lista de estados habilitados (*enabledEventStates*), a função de transição de estado (*MakeTransition*) e a função de execução de estado (*Loop*).
    - O destrutor também está presente, mas não possui implementação adicional.
2. **Métodos set e get:**
    - Os métodos *setEvent(long event)* e *setActualState(int state)* são implementados aqui. Eles atribuem os valores fornecidos para *actualEvent* e *actualState*, respectivamente.
    - Os métodos *getActualState()*, *getNumStates()* e *getEnabledEvent()* são implementados retornando os valores solicitado.
3. **Método getEnabledEvent():** O método *getEnabledEvent()* agora retorna o valor de *enabledEventStates* correspondente ao estado atual (*actualState*). Essa função permite obter o valor dos eventos habilitados para o estado atual do automato.

Existem também comentários que indicam a intenção de adicionar outras funcionalidades, essa inserção ocorre apos a execução da função *ConvertDEStoINO()*.

5.3.3 `UserFunctionsDefault.cpp`

Neste arquivo existe a adição de algumas implementações e funcionalidades adicionais relacionadas aos eventos controláveis e incontroláveis, bem como ações de estado.

1. **Função *long getEventControllablble()*:**
    - Essa função verifica se há dados disponíveis na porta Serial (*Serial.available()*). Caso não haja dados disponíveis, ela retorna 0, indicando que não há eventos controláveis.
    - Se houver dados disponíveis, ela usa *Serial.parseInt()* para ler um valor numérico da porta Serial e armazena esse valor na variável *valorSerial*.
    - Em seguida, o valor *valorSerial* é retornado como o evento controlável obtido. Isso permite a leitura de eventos a partir da porta Serial.
2. **Função *void setupPin()*:** Essa função está presente, mas não possui implementação adicional em relação ao que foi declarado no arquivo de cabeçalho, deixando ela aberta para a configuração pelo usuário final.

Existem comentários indicando a intenção de adicionar uma função *getEventUncontrollable()* e ações de estado, mas essas implementações não estão presentes no código ainda.

5.4 Estudo de caso

Nesta seção, será apresentado um estudo de caso que ilustrará a utilização da solução desenvolvida, com o objetivo de demonstrar a eficiência do projeto. A metodologia para esse estudo de caso pode ser ilustrada pela Figura 10

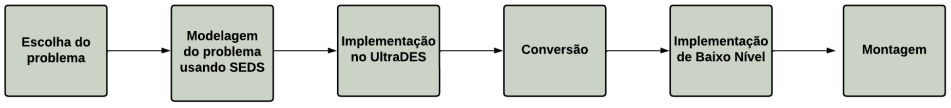

Figura 10 – Passo a passo para a implementação do estudo de caso

O estudo de caso abordará todas as etapas, desde a modelagem dos SEDs utilizando o UltraDES até o resultado final com as alterações relacionadas à implementação de baixo nível.

5.4.1 Escolha do problema: Pequena Fábrica

O sistema que será trabalhado baseia-se na Pequena Fábrica, também conhecida como *Small Factory*, que é uma planta bem conhecida na área de Sistemas a Eventos Discretos, sendo utilizada em (WONHAM; CAI, 2017). A planta é composta de duas máquinas conectadas por um *buffer* unitário, como pode ser visto na Figura 11. A primeira máquina trabalha no produto e deposita no *buffer*, a segunda máquina retira o produto do *buffer* e realiza outro trabalho sobre ele. Em geral, a especificação de segurança da Pequena Fábrica é garantir que não haja o *underflow* nem o *overflow* do *buffer*, ou seja, garantir que a primeira máquina não coloque um produto no *buffer* se ele já está cheio e que a segunda máquina não tente retirar um produto do *buffer* se este está vazio.

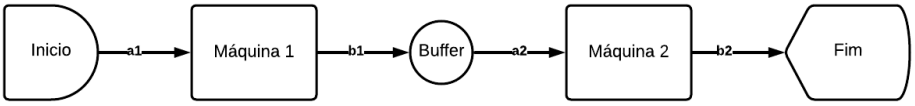

Figura 11 – Diagrama da Pequena Fábrica

5.4.2 Modelagem usando SED

Definindo os eventos α_1 e α_2 como eventos controláveis representando o comando de início das máquinas $M1$ e $M2$, enquanto os eventos β_1 e β_2 modelam os eventos não controláveis pois representam os sinais de fim da operação das máquinas $M1$ e $M2$, respectivamente.

Com base nessas informações pode-se encontrar um modelo para essa planta com base nos autômatos para as máquinas 1 e 2 como visto na Figura 12

(a) M1- Máquina 1 (b) M2- Máquina 2

Figura 12 – Modelo das plantas para a pequena fábrica

Para a máquina M_1 considera-se uma tarefa completa o ato de retirar uma peça e depositá-la no *buffer*. Já para a máquina M_2 uma tarefa completa consiste no ato de retirar uma peça do *buffer* e voltar ao seu estado original. Por isso marcam-se os estados D_1 e D_2 das duas plantas.

Já a especificação de segurança representa a restrição imposta pela presença do *buffer* e pode ser modelada como se encontra na Figura 13

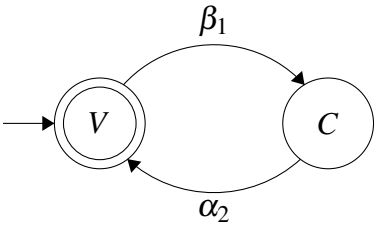

Figura 13 – Modelo da especificação para a pequena fábrica

Em que, *V* representa o estado em que o *buffer* está vazio enquanto *C* representa o estado em que o *buffer* está cheio. É interessante notar que o *buffer* fica cheio quando a máquina 1 termina seu processo, ou seja, quando o evento β_1 ocorre e fica vazio quando o máquina dois começa seu processamento, o que é indicado pela ocorrência de α_2.

Para se entender o modelo de malha aberta do sistema basta fazer a composição síncrona $G = M1||M2$ que resulta no automato presente na Figura 14. É fácil perceber que a planta *G* modela comportamentos que não respeitam os requisitos citados e por esse motivo se faz necessário o controle da mesma, para que assim seja possível atender a especificação modelada.

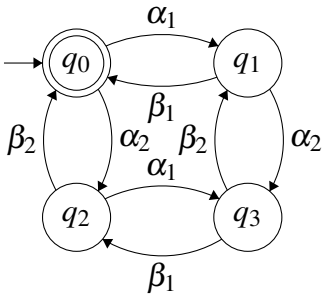

Figura 14 – Composição síncrona das plantas

Em que:

- q_0 representa o estado em que ambas as máquinas estão desligadas;
- q_1 representa o estado em que a máquina 1 está ligada e a máquina 2 está desligada;
- q_2 representa o estado em que a máquina 2 está ligada e a máquina 1 está desligada;
- q_3 representa o estado em que ambas as máquinas estão ligadas;

Para essa implementação física da prova de conceito, foi desenvolvido um sistema de transporte com duas esteiras com base na modelagem da pequena fábrica apresentada. Cada uma das esteiras representa as máquinas M1 e M2, e o *buffer* é representado por uma mesa localizada entre as esteiras.

5.4.3 Implementação no UltraDES

Com base nessa analise e utilizando o UltraDES modelou-se esse sistema da forma descrita no algorítimo 8

```
// Estados da Maquina 1
State S11 = new State("S11", Marking.Marked);
State S12 = new State("S12", Marking.Unmarked);

// Estados da Maquina 2
State S21 = new State("S21", Marking.Marked);
State S22 = new State("S22", Marking.Unmarked);

// Estados do Buffer com uma posicao
State SB1 = new State("Empty", Marking.Marked);
State SB2 = new State("Full", Marking.Unmarked);

// Eventos controlaveis
Event a1 = new Event("a1", Controllability.Controllable);
Event a2 = new Event("a2", Controllability.Controllable);

// Eventos nao controlaveis
Event b1 = new Event("b1", Controllability.Uncontrollable);
Event b2 = new Event("b2", Controllability.Uncontrollable);

// Automato Maquina 1
var G1 = new DeterministicFiniteAutomaton(new[]{
    new Transition(S11, a1, S12),
    new Transition(S12, b1, S11)
  }, S11, "G1");

// Automato Maquina 2
var G2 = new DeterministicFiniteAutomaton(new[]{
    new Transition(S21, a2, SB2),
    new Transition(SB2, b2, S21)
  }, S21, "G2");

// Automato Buffer (Especificacao)
var E = new DeterministicFiniteAutomaton(new[]{
    new Transition(SB1, b1, SB2),
    new Transition(SB2, a2, SB1)
  }, SB1, "E");

var Planta = G1.ParallelCompositionWith(G2);

var Supervisor = DeterministicFiniteAutomaton.MonolithicSupervisor(
        new[] { G1, G2 },
        new[] { E },
        true
            );
```

Algoritmo 8: Exemplo UltraDES : Pequena fabrica

Utilizou-se também o UltraDES para chegar ao supervisor do sistema utilizando da abordagem monolítica. Esse supervisor, presente na Figura 15, irá garantir que o funcionamento do sistema siga a especificação.

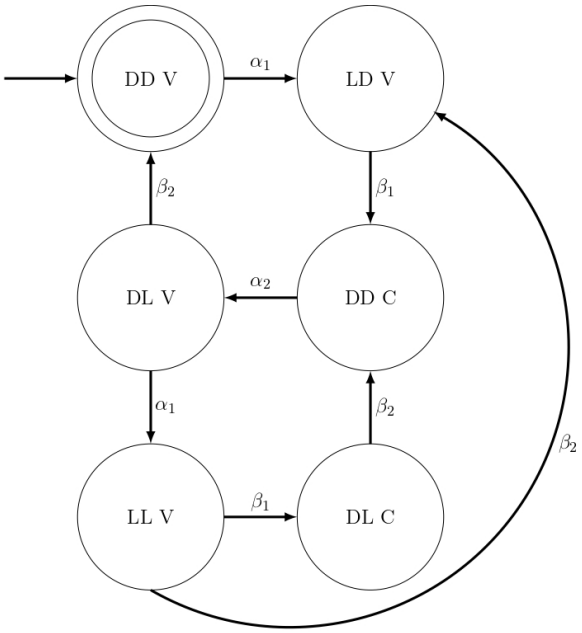

Figura 15 – Supervisor para a Pequena Fábrica

5.4.4 Conversão

Após a modelagem do problema utilizando o UltraDES, é criada uma lista de autômatos utilizando o objeto em *C# List < DeterministicFiniteAutomaton >*. Nessa lista, são adicionados os autômatos correspondentes à máquina 1, à máquina 2 e ao supervisor. Por fim, é chamada a função *ConvertDEStoINO*, responsável por realizar a conversão do modelo em UltraDES para o código compatível com a plataforma Arduino em *C++*, como pode ser visto no algorítimo 9

```
1 List<DeterministicFiniteAutomaton> ListaEstudoCaso= new List<
      DeterministicFiniteAutomaton>();
2
3
4 ListaEstudoCaso.add(Supervisor);
5 ListaEstudoCaso.add(G1);
6 ListaEstudoCaso.add(G2);
7
8 INOGenerator.ConvertDEStoINO(ListaEstudoCaso);
```

Algoritmo 9: Chamada da função de conversão

Para melhor visualizar o resultado dessa conversão passa-se agora a alguns dos códigos resultantes dessa operação.

5.4.4.1 Criando a lista de autômatos

Durante essa etapa, foram coletadas as informações necessárias para inicializar objetos do tipo *automato*, como ilustrado na Figura 7. Esses objetos são então instanciados e armazenados em um vetor, no exemplo trabalhado da pequena fabrica essa etapa pode ser observada no Algoritmo 10. A criação dessa lista de autômatos permite uma organização e gerenciamento eficientes do projeto resultante pois ao armazenar esses objetos em um vetor, é possível acessá-los e manipulá-los de forma continua durante a execução do programa.

```
// Create a vector to store the automatons
Automaton automata[NUMBER_AUTOMATON] = {
Automaton(6,enabledEventStatesAutomaton0,
&MakeTransitionAutomaton0,&Automaton0Loop),

Automaton(2,enabledEventStatesAutomaton1,
&MakeTransitionAutomaton1,&Automaton1Loop),

Automaton(2,enabledEventStatesAutomaton2,
&MakeTransitionAutomaton2,&Automaton2Loop)};
```

Algoritmo 10: Criando a lista de autômatos

5.4.4.2 Representação da transição

Considerando a máquina 1 que foi modelada no primeiro automato da Figura 12 observar-se a seguinte função de transição.

```
int MakeTransitionAutomaton1(int State, long Event) {
if(Event==0){return State;}
if (State == 0 && (Event & 0b0010)>0){ return 1;}
if (State == 1 && (Event & 0b1000)>0){ return 0;}
return(State);}
```

Algoritmo 11: Função de transição do Motor 1

Essa função verifica se o evento é zero e, nesse caso, retorna o estado atual. Em seguida, a função realiza verificações condicionais com base no estado atual e no evento ocorrido, retornando o novo estado correspondente à transição. Se nenhuma condição é satisfeita, o estado atual é retornado, indicando que não houve transição.

5.4.4.3 Representação dos Estados

Para ilustrar a representação do estado no modelo convertido usando o exemplo da pequena fábrica e focando no estado marcado D_1, que representa que a mesma está desligada, da máquina M_1, pode-se considerar que o estado é representado pelos seguintes aspectos:

- Função de Ação do Estado: A função de ação do estado D_1 da máquina M_1 descreve as ações a serem executadas quando o sistema está nesse estado. Essa ação é chamada por meio da função de loop do automato que é dada da seguinte forma:

```
1 void Automaton1Loop(int State){
2   ActionAutomatons1[State]();
3 }
```

Algoritmo 12: Ação de estado: Motor 1 ligado

- Eventos Habilitados do Estado: Os eventos habilitados do estado D_1 são os eventos que podem ocorrer quando o sistema está nesse estado, na representação pode-se encontrar esses eventos através do vetor de eventos habilitados, Por exemplo:

```
1 long enabledEventStatesAutomaton1[2]={ 0b0010, 0b1000};
```

Algoritmo 13: Vetor de Eventos habilitados para $M1$

Observando $0b1000$ percebe-se que somente o evento da quarta posição está habilitado para o estado D_2, esse evento é exatamente o evento relacionado à ação que desligará a máquina 1.

- Número do Estado: Cada estado no modelo convertido possui um número único que o identifica. Essa é uma identificação lógica tendo em vista que não existe uma atribuição desse valor ao estado. No exemplo trabalhado o estado D_1 da máquina M_1 pode ser identificado pelo número 1, isso é observado no nome da função de ação de estado como visto no algorítimo 12

5.4.5 Implementação de Baixo Nível

Durante o processo de implementação em baixo nível do exemplo da pequena fábrica, foram realizadas as seguintes alterações no arquivo `UserFunctions.cpp`, com o objetivo de garantir o correto funcionamento do sistema de transporte automatizado.

5.4.5.1 Configuração dos pinos

Inicialmente, foi considerada a configuração de pinos presente no algorítimo 14.

```
#define OUTM1 5
#define OUTM2 4
#define EVENTIN1 16
#define EVENTIN2 15

void setupPin(){
    pinMode(OUTM1,OUTPUT);
    pinMode(OUTM2,OUTPUT);
    pinMode(EVENTIN1,INPUT);
    pinMode(EVENTIN2,INPUT);
}
```

Algoritmo 14: Configuração dos pinos

Nessa configuração, os pinos 5 e 4 são responsáveis pela ativação das máquinas 1 e 2, respectivamente. Já os pinos 16 e 15 são utilizados para receber os eventos não controláveis, os quais são provenientes da dinâmica física do sistema.

5.4.5.2 Recebimento dos eventos não controláveis

Os eventos não controláveis foram implementados pelo usuario final, da seguinte maneira: a função "*EventUncontrollableb1*" e "*EventUncontrollableb2*" que verifica o estado de um determinado pino de entrad. Se o valor lido nesse pino for igual a 0, a função retorna "*true*", indicando que o evento não controlável ocorreu. Caso contrário, a função retorna "*false*", indicando que o evento não controlável não ocorreu. Como pode ser visto no algorítimo 15.

```
bool EventUncontrollableb1(){
  if(digitalRead(EVENTIN1) == 0){return true;}
  return false;}

bool EventUncontrollableb2(){
  if(digitalRead(EVENTIN2) == 0){return true;}
  return false;}
```

Algoritmo 15: Implementação dos eventos não controláveis

5.4.5.3 Implementação das ações de estados

As funções mencionadas nessa seção foram implementadas para controlar o estado das máquinas 1 e 2, como é visto no algorítimo 16.

```
void StateActionAutomaton1State0(){digitalWrite(OUTM1, LOW);}

void StateActionAutomaton1State1(){digitalWrite(OUTM1, HIGH);}

void StateActionAutomaton2State0(){digitalWrite(OUTM2, LOW);}

void StateActionAutomaton2State1(){digitalWrite(OUTM2, HIGH);}
```

Algoritmo 16: Implementação das ações de estados

A função *StateActionAutomaton1State0* desativa a máquina 1, configurando o pino de saída *OUTM1* como *LOW*. Em contrapartida, a função *StateActionAutomaton1State1* ativa a máquina 1, configurando o pino de saída *OUTM1* como *HIGH*. Da mesma forma, a função *StateActionAutomaton2State0* desliga a máquina 2, definindo o pino de saída *OUTM2* como *LOW*, enquanto a função *StateActionAutomaton2State1* liga a máquina 2, configurando o pino de saída *OUTM2* como *HIGH*. Para realizar essas ações, todas as funções utilizam a instrução *digitalWrite* para controlar o estado dos respectivos pinos de saída. Essas implementações garantem o correto funcionamento do sistema de transporte automatizado, permitindo a ativação e desativação das máquinas conforme necessário.

5.4.6 Montagem

Para visualizar e representar essas esteiras, foi criado um modelo em 3D para cada uma delas. A Figura 16 ilustra esses modelos, permitindo uma melhor compreensão do sistema físico implementado.

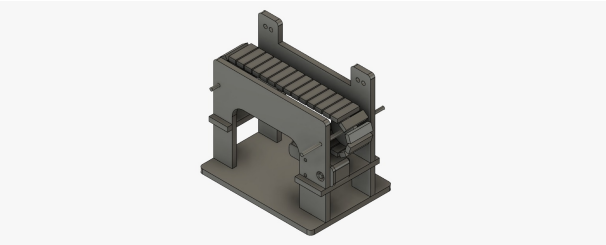

Figura 16 – Modelo 3D das esteiras do sistema

Após a criação do modelo em 3D das esteiras, foi possível utilizar a máquina de corte a laser para produzir a estrutura física do sistema de transporte. O resultado dessa etapa pode ser observado na Figura 17, onde a estrutura final das esteiras é apresentada.

A utilização da máquina de corte a laser proporcionou a obtenção de uma estrutura resistente e precisa, que atende aos requisitos de montagem e funcionamento do sistema de transporte visando a durabilidade do projeto.

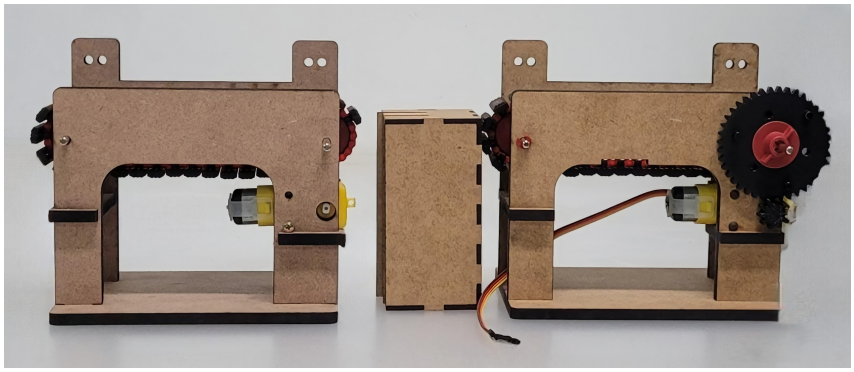

Figura 17 – Montagem final das esteiras do sistema de transporte

Nessa montagem especifica foi utilizado o microcontrolador *NodeMCU*, que possui semelhanças com os microcontroladores Arduino. Essa escolha permite que a solução desenvolvida seja adaptada tanto para o *NodeMCU* quanto para os microcontroladores Arduino.

A Figura 18 ilustra o diagrama esquemático do circuito de ativação das esteiras utilizando o relé e o *driver ULN2003A*. Esse circuito é responsável por conectar o microcontrolador *NodeMCU* (ou Arduino) aos componentes de acionamento das esteiras, permitindo que o sistema de controle automatizado seja efetivamente implementado.

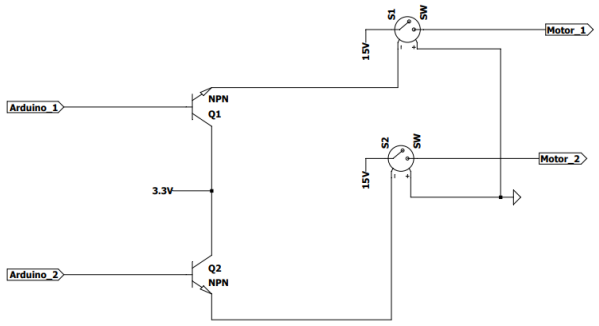

Figura 18 – Circuito de controle

A utilização de relés no circuito de ativação das esteiras é uma prática comum para separar o circuito de controle de baixa tensão do circuito de carga das esteiras, que geralmente requerem uma tensão mais elevada para seu acionamento. Essa separação garante a segurança e eficiência no acionamento das esteiras.

O *driver ULN2003A* é um componente adequado para essa finalidade, pois possui várias saídas capazes de controlar os relés de forma independente. Ele oferece proteção contra sobretensão e possui capacidade de acionamento de correntes mais altas, o que o torna ideal para controlar as esteiras.

5.4.7 Resultado

Após o desenvolvimento e realização de testes do programa gerado pela solução proposta, o próximo passo foi realizar o *upload* desse programa para o módulo *NodeMCU* (ou Arduino). O processo de *upload* foi realizado com sucesso, e o programa executou de maneira satisfatória, obedecendo ao comportamento definido pelo supervisor encontrado utilizando o UltraDES. Isso demonstra que a solução implementada é capaz de controlar adequadamente o sistema de acordo com as especificações estabelecidas, garantindo um funcionamento correto e confiável.

5.4.8 Conclusão

Em conclusão, o estudo de caso realizado comprova a eficiência da solução desenvolvida para implementação de Sistemas a Eventos Discretos (SED) utilizando a biblioteca Ul-

traDES e o Arduino. A conversão do comportamento modelado para o código Arduino foi bem-sucedida, proporcionando uma implementação mais simples e acessível.

A solução proposta simplifica significativamente a tarefa de programar a lógica de automação, eliminando a necessidade de escrever todo o código manualmente. Os resultados positivos obtidos na prova de conceito demonstram a viabilidade e o potencial da solução para a implementação de sistemas de controle automatizados em microcontroladores.

Com base nessa validação, é possível avançar para aplicações mais complexas e escaláveis, utilizando a mesma metodologia e princípios desenvolvidos neste projeto. A solução apresentada oferece uma abordagem promissora para simplificar e agilizar o desenvolvimento de sistemas de automação, permitindo que pesquisadores e desenvolvedores foquem em aspectos mais específicos de seus projetos, sem se preocupar com a implementação detalhada da lógica de automação.

6 Conclusão

Em conclusão, este trabalho apresentou uma solução para automatizar a implementação de sistemas a eventos discretos utilizando a conversão de autômatos finitos modelados pela biblioteca UltraDES em código compatíveis com a plataforma Arduino. A abordagem proposta mostrou-se viável e eficiente, permitindo a tradução da lógica de automação de forma prática e acessível, permitindo sua implementação em projetos acadêmicos do LACSED. O projeto demonstrou a viabilidade e utilidade dessa abordagem proporcionando uma forma mais acessível e intuitiva de desenvolver soluções de automação baseadas em SEDs. Além disso, a disponibilização da biblioteca UltraDES e a documentação detalhada da metodologia empregada permitem que a comunidade acadêmica tenha acesso a uma ferramenta poderosa para o desenvolvimento de seus próprios projetos usando a teoria de controle supervisório. Isso contribui para o avanço das pesquisas no campo de eventos discretos, permitindo aos pesquisadores concentrarem-se em aspectos mais específicos de seus projetos, sem a necessidade de se preocuparem com a implementação detalhada da lógica de automação.

Para projetos futuros, há possibilidade de explorar melhorias e expansões no escopo do trabalho atual. Uma opção é a integração com outros dispositivos e tecnologias, possibilitando a automação de sistemas mais complexos. Além disso, pode-se investir no desenvolvimento de um supervisório ou planejador que se comunique com a placa Arduino, enviando uma lista de eventos controláveis.

Outras ideias para projetos futuros incluem a integração de tecnologias de comunicação, como Bluetooth ou Wi-Fi, para permitir o controle e monitoramento remoto dos dispositivos automatizados, oferecendo maior conveniência e flexibilidade aos usuários. Essas iniciativas proporcionariam maior flexibilidade e eficiência aos usuários, ao mesmo tempo em que ampliariam as capacidades do sistema desenvolvido.

Referências

ALVES, L.; MARTINS, L.; PENA, P. Ultrades - a library for modeling, analysis and control of discrete event systems. *IFAC-PapersOnLine*, v. 50, p. 5831–5836, 07 2017. 23

BARRAGáN, H. *Wiring*. August 14, 2020. Wiring. Acessado em 15 de maio de 2023. 24

CASSANDRAS, C.; LAFORTUNE, S. *Introduction to Discrete Event Systems*. [S.l.: s.n.], 2010. 800 p. ISBN 1441941193. 16, 17, 20

FABIAN, M.; HELLGREN, A. Plc-based implementation of supervisory control for discrete event systems. In: . [S.l.: s.n.], 1998. v. 3, p. 3305 – 3310 vol.3. 26

LOPES, Y. et al. Local modular supervisory implementation in microcontroller. In: . [S.l.: s.n.], 2012. 30

PENA, P. N. *Verificação de conflito na supervisão de sistemas concorrentes usando abstrações*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2007. 21, 22

QUEIROZ, M. Hering de; CURY, J. Modular control of composed systems. In: . [S.l.: s.n.], 2000. v. 6, p. 4051 – 4055 vol.6. ISBN 0-7803-5519-9. 22

QUEIROZ, M. Hering de; CURY, J. Synthesis and implementation of local modular supervisory control for a manufacturing cell. In: . [S.l.: s.n.], 2002. p. 377 – 382. ISBN 0-7695-1683-1. 7, 27, 28

RAHMAN, S.; ABU-GHAZALEH, N.; NAJJAR, W. Pdes-a: Accelerators for parallel discrete event simulation implemented on fpgas. *ACM Transactions on Modeling and Computer Simulation*, v. 29, p. 1–25, 04 2019. 29

RU, Y.; HADJICOSTIS, C. N. Fault diagnosis in discrete event systems modeled by partially observed petri nets. *Discrete event dynamic systems*, Springer US, Boston, v. 19, n. 4, p. 551–575, 2009. ISSN 0924-6703. 16

SKOLDSTAM, M.; AKESSON, K.; FABIAN, M. Modeling of discrete event systems using finite automata with variables. p. 3387–3392, 2007. 16

TEAM, T. A. *Getting started with Arduino products*. August 14, 2020. Arduino. Acessado em 15 de maio de 2023. 24

VIEIRA, A. D. et al. A method for plc implementation of supervisory control of discrete event systems. *IEEE Transactions on Control Systems Technology*, v. 25, p. 175–191, 2017. 28

WONHAM, W.; CAI, K. *Supervisory Control of Discrete-Event Systems, v.20170901*. [S.l.: s.n.], 2017. 20, 50

WONHAM, W.; RAMADGE, P. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems*, Springer London, v. 1, n. 1, p. 13–30, fev. 1988. ISSN 0932-4194. 21