Geração Automática de Código para Arduino com base na Teoria de Controle Supervisório integrado com o UltraDES

Matheus P. Loures * **Lucas V. R Alves** **

* *Programa de Pós-Graduação em Engenharia Elétrica - Universidade Federal de Minas Gerais - Av. Antônio Carlos 6627,31270-901, Belo Horizonte,MG, Brasil,(e-mail: mploures@ufmg.br).*
** *Colégio Técnico, Universidade Federal de Minas Gerais, Brasil, (e-mail: lucasvra@ufmg.br).*

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 at LACSED (Laboratory for Analysis and Control of Discrete Event Systems). 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.

Resumo: O principal objetivo deste trabalho é desenvolver um programa capaz de automatizar o processo de aplicação de sistemas de eventos discretos a sistemas embarcados. Para alcançar esse fim, utiliza-se a implementação de autômatos finitos provenientes da biblioteca UltraDES, os quais são convertidos em código Arduino. Esta abordagem visa facilitar a implementação de soluções de automação em pesquisas acadêmicas conduzidas no LACSED (Laboratório de Análise e Controle a Sistemas de Eventos Discretos). Os resultados obtidos demonstram a eficácia da solução desenvolvida. A implementação de autômatos finitos em código Arduino possibilita a automação dos processos de sistemas de eventos discretos.

Keywords: Discrete event systems, Supervisory control theory, Modeling, Embedded systems, Functional programming

Palavras-chaves: Sistemas de eventos discretos, Teoria de controle supervisório, Modelagem, Sistemas embarcados, Programação funcional

1. INTRODUÇÃO

As demandas atuais por otimização, personalização e automação nas novas metodologias de produção desafiam as estratégias do controle clássico, que se baseiam na modelagem usando equações diferenciais e de diferença, devido à interação distinta desses sistemas com o ambiente, percebendo-o por meio de eventos ou estímulos instantâneos. Esses sistemas, conhecidos como Sistemas a Eventos Discretos (SEDs), estão presentes em diversas aplicações contemporâneas, como automação de manufatura, redes de computadores, logística de cadeia de produção e robótica.

Existe uma grande demanda em diversas linhas de pesquisa no campo da Teoria de Controle Supervisório por simulação tanto por meio de *softwares* quanto a implementação desses estudos em plantas físicas didáticas, visando uma melhor visualização dos resultados obtidos.

Contudo a implementação de SEDs em sistemas embarcados requer a criação de um novo programa para cada autômato desenvolvido, o que consome tempo. Para superar esse obstáculo, é importante desenvolver uma aplicação que converta os modelos de autômatos finitos já modelados em código *C++* compatível com o sistemas embarcados. Isso permitirá a aplicação da Teoria de Controle Supervisório de Sistemas a Eventos Discretos em plataformas de *hardware* como o *Arduino*, sendo esse o objetivo deste trabalho.

O artigo está organizado da seguinte forma: após a introdução, que oferece um panorama sobre o assunto, a seção 2 abordará os conceitos fundamentais necessários para o desenvolvimento do projeto. Em seguida, a seção 3 discutirá a arquitetura desenvolvida explicando as escolhas de projeto feitas com o objetivo de produzir uma solução robusta. Posteriormente, na seção 4, um estudo de caso para mostrar os resultados obtidos será apresentado. Por fim, na seção 5, serão apresentadas as conclusões do trabalho.

2. PRELIMINARES

Os sistemas dinâmicos evoluem seus estados internos seguindo um conjunto de regras definidas, descrevendo a dinâmica das variáveis, suas interações e a comunicação com o meio por sinais de entrada e saída. Os Sistemas a Eventos Discretos (SEDs) são uma classe dos sistemas dinâmicos que apresenta transições abruptas entre estados em momentos específicos, chamados de eventos, que abstraem fenômenos espontâneos, ações específicas ou condições simultâneas. A modelagem desses sistemas pode ser feita com técnicas como linguagens e autômatos, redes de Petri e outras representações. Além disso, para esse tipo de sistemas, a ocorrência simultânea de dois

ou mais eventos não é permitida (Cassandras and Lafortune, 2010).

2.1 Linguagens e Autômatos Finitos Determinísticos

Em um SED, os eventos são representados como símbolos pertencentes a um alfabeto,Σ, e as sequências de eventos como palavras em uma linguagem, o conjunto de todas as sequências finitas de um alfabeto é representado por Σ^*. Expressões regulares são usadas para representar linguagens de forma concisa, permitindo 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, e as expressões regulares são compostas por cadeias e operações aplicadas a essas cadeias (Cassandras and Lafortune, 2010).

As expressões regulares podem representar todas as linguagens regulares, que podem ser geradas por autômatos. Autômatos são grafos orientados onde os vértices representam os estados do sistema e as arestas representam as transições ativadas por eventos. Esses autômatos permitem modelar o comportamento de sistemas a eventos discretos formalmente.

Definição 1. Um autômato finito determinístico é definido por uma quíntupla $G = (Q, \Sigma, \delta, q_0, Q_m)$. Em que Q é um conjunto finito de estados, Σ é o conjunto de eventos ou alfabeto de G, δ ($\delta : Q \times \Sigma \to Q$) é a função de transição de estados, $q_0 \in Q$ é o estado inicial e $Q_m \subseteq Q$ é o conjunto de estados marcados de G

A função de transição δ pode ser estendida para cadeias 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 $s \in \Sigma^*$.

2.2 Teoria de Controle Supervisórios

A Teoria de Controle Supervisório (TCS) (Wonham and Cai, 2017), fundamentada na teoria dos Autômatos Finitos Determinísticos (AFD), é empregada para modelar sistemas complexos e prevenir bloqueios. O objetivo é encontrar um supervisor não bloqueante e minimamente restritivo que implemente uma especificação pela desabilitação de um subconjunto de eventos e evitando sequências de eventos inaceitáveis.

Considerando que $\Sigma = \Sigma_c \cup \Sigma_u$, em que Σ_c representa eventos controláveis e Σ_u eventos não controláveis. O supervisor trabalha desabilitando eventos de Σ_c para fazer com que a planta assuma o comportamento desejado $K = L(G)||E$.O controle supervisório visa 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, obtendo assim o comportamento desejado para o sistema em malha fechada.(Cassandras and Lafortune, 2010):

- **Controle Monolítico:** Nesta abordagem de TCS, uma única planta, modelada completamente ou por composição de sub-plantas modeladas por AFDs.(Wonham and Cai, 2017) O controle adiciona uma estrutura que controla a linguagem gerada pelo sistema dentro de limites.
- **Controle Modular:** O controle modular, proposto em (Wonham and Ramadge, 1988), combina ações de supervisores para resolver o problema original, desabilitando eventos que são desabilitados por pelo menos um supervisor. A modularidade é verificada garantindo que todo prefixo de uma cadeia marcada nos supervisores seja prefixo de pelo menos uma cadeia na interseção das linguagens dos supervisores.
- **Controle Modular Local:** Uma extensão da técnica modular é o controle modular local (Hering de Queiroz and Cury, 2000) que cria um supervisor para cada especificação de controle, considerando apenas os subsistemas afetados. Cada supervisor local tem uma visão parcial da planta, permitindo uma abordagem descentralizada. É importante verificar se os supervisores locais não são conflitantes garantindo que suas ações em conjunto não são bloqueantes

2.3 UltraDES

O UltraDES (Alves et al., 2017) é uma biblioteca em C# construída sobre o *.NET Framework*, oferecendo estruturas e algoritmos para modelagem, análise e controle de Sistemas de Eventos Discretos (SEDs). Compatível com qualquer linguagem de programação e plataforma que suporte *.NET Standard* 2.0, incluindo ambientes móveis como Android e *iOS*, a biblioteca simplifica o desenvolvimento e validação de métodos para SEDs. Com paradigmas de programação orientada a objetos e funcional, o UltraDES oferece estruturas para eventos, estados e autômatos, bem como implementações de algoritmos relevantes na Teoria do Controle Supervisório, e estruturas de dados para grafos e redes de Petri.

2.4 Implementações de Controle Supervisório

A implementação de SED's em Controladores Lógicos Programáveis (CLPs) é uma estrategia já bem estabelecida na literatura. Esse processo envolve a modelagem da planta e a codificação do comportamento de uma máquina de estados finitos na linguagem *ladder*. Um exemplo prático é o trabalho de Fabian and Hellgren (1998), que ilustra essa metodologia de maneira bem detalhada. Além disso, o estudo de Hering de Queiroz and Cury (2002) aplica a Teoria de Controle Supervisório em uma célula de manufatura controlada por um CLP, propondo uma metodologia que emprega supervisores reduzidos, implementados em uma estrutura de três níveis como visto na Figura 1. Este método visa coordenar múltiplos subsistemas na célula de manufatura, assegurando eficiência e segurança no controle de processos industriais.

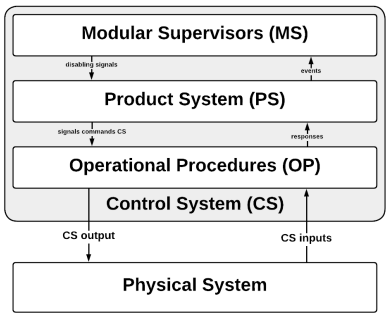


Figura 1. Arquitetura de Controle Supervisório Hering de Queiroz and Cury (2002)

3. METODOLOGIA

Nesta seção, são abordadas a organização e as decisões relacionadas à implementação de baixo nível, explicando como as estruturas principais de um Sistema a Eventos Discretos (SED) foram modeladas para o *Arduino*. Também são apresentadas as decisões de implementação e a arquitetura escolhida para o fluxo principal no *Arduino*, com o objetivo de oferecer ao

leitor uma compreensão clara do desenvolvimento e da implementação do sistema de controle supervisório no *Arduino*, contribuindo para a análise crítica do projeto.

3.1 Tradução das Estruturas de SEDs

Para adaptar a lógica de um SED para o *Arduino*, é crucial representar estados, eventos, transições e autômatos de forma eficiente, considerando as limitações de memória e processamento do dispositivo. Nessa seção serão relatados as escolhas realizadas na busca de garantir a execução correta da aplicação, mesmo em dispositivos com recursos limitados, mantendo a flexibilidade e adaptabilidade do sistema para diferentes cenários e necessidades específicas do usuário.

- **Estados:** Cada estado é representado por um número inteiro de 16 bits, permitindo até 65.535 estados. Cada estado possui uma função associada que executa tarefas específicas relacionadas a esse estado. Essa abordagem permite a execução da lógica do sistema de forma eficiente e ocupando menos espaço na memória, além de facilitar a programação e compreensão do código.
- **Eventos:** Os eventos são representados por uma estrutura que encapsula um *array* de `uint8_t`, onde cada bit indica se o evento ocorreu ou não, o tamanho do *array* é calculado dinamicamente com base no número de eventos. Operações de máscara de bits como *AND*, *OR*, *XOR* e *NOT* são utilizadas para manipular os eventos de forma eficiente. Os eventos controláveis são obtidos por um sinal externo ou por uma sequência de controle fixada em código, enquanto os eventos não controláveis são adquiridos por uma função que verifica fenômenos da planta. As transições de estado são determinadas com base nos eventos ocorridos.
- **Transição:** Uma transição em um Sistema a Eventos Discretos (SED) representa a mudança de um estado atual para outro, refletindo as consequências da ocorrência de um evento. No *Arduino*, essa transição é implementada por meio de uma função que recebe o estado atual e o último evento ocorrido, determinando o próximo estado com base nas transições definidas no autômato.
- **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. A classe possui as seguintes funções:
  - `Automaton()`: Construtor da classe, recebe número de estados, lista de eventos habilitados por estado, função de transição e função de execução.
  - `~Automaton()`: Destrutor, libera memória alocada.
  - `getActualState()`: Retorna o estado atual do autômato.
  - `getNumStates()`: Retorna o número total de estados.
  - `getEnabledEvent()`: Retorna os eventos habilitados no estado atual.
  - `setActualState(int state)`: Define o estado atual do autômato.
  - `(*MakeTransition)(int state, long event)`: Ponteiro para a função de transição de estados.
  - `(*Loop)(int state)`: Ponteiro para a função de loop, chamando ações de estado com base no estado atual.

3.2 Estrutura do Projeto

No desenvolvimento do projeto, foi dada prioridade à eficiência, flexibilidade e facilidade de manutenção. Optou-se por armazenar os códigos base necessários para a conversão das implementações de Sistemas a Eventos Discretos (SEDs) em sistemas embarcados em sete arquivos separados, proporcionando organização e mantendo o código-fonte mais limpo e legível. Além disso, a separação dos códigos torna mais claro o processo de conversão e facilita a compreensão do programa como um todo, como observado na Figura 2.

`INOGenerator.cs`: O arquivo `INOGenerator.cs` em C# desempenha um papel fundamental no projeto, pois é o responsável por traduzir a lógica dos autômatos do UltraDES para uma forma compatível com o Arduino. Nele os arquivos são lidos e por fim adiciona-se a lógica dos autômatos, conforme mostrado na Figura 2. De modo que:

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

`mainDefault.INO`: Coordena a execução do programa final, iniciando a lógica necessária e chamando as funções de acordo com a lógica dos autômatos. Depois da conversão, o `main.INO` assegura o correto fluxo de execução e interação entre os componentes do sistema, com poucas alterações, devido à sua estrutura reutilizável.

`AutomatonDefault.h`: Arquivo de cabeçalho onde são declaradas as estruturas, classes e funções utilizadas no projeto, organizando as declarações para facilitar a leitura e manutenção do código. A classe "`Automaton`" e suas funções fornecem uma estrutura reutilizável e consistente, com funções de transição de estado e de ação adicionadas dinamicamente para adaptar-se às necessidades específicas de cada autômato.

`AutomatonDefault.cpp`: Implementa as funções de *loop* e de transição de estados para cada autômato, projetado para não requerer alterações diretas pelo usuário final, garantindo estabilidade e consistência, e permitindo que o usuário se concentre na personalização de outros aspectos do sistema.

`UserFunctionsDefault.cpp`: Contém a implementação das funções que verificam a ocorrência dos eventos não controláveis e controláveis, inserindo-os em um vetor de eventos e as ações dos estados. Essas implementações podem ser modificadas pelo usuário final para personalizar o sistema de acordo com as necessidades específicas do projeto.

`EventDefault.h`: Implementa e gerencia a estrutura *Event*, que recebe e encapsula os eventos do sistema, estabelecendo o número de eventos do sistema para um gerenciamento eficiente de memória em sistemas embarcados. Desenvolve as principais operações sobre eventos, consistindo em operações binárias que garantem que o sistema possa lidar com eventos de forma flexível e eficiente, utilizando apenas a memória interna disponível no sistema embarcado.

3.3 Arquitetura da Função de Tradução

O processo de tradução de um SED modelado no *Ultrades* para a implementação no *Arduino*, considerando as limitações de

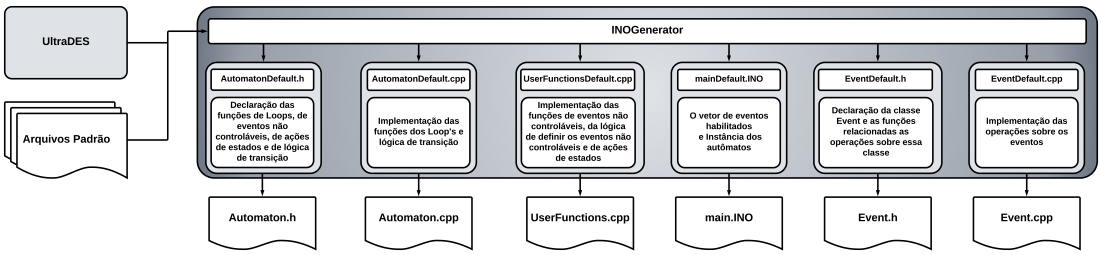


Figura 2. Estrutura dos arquivos da solução

memória e processamento do Arduino, pode ser descrito pelas etapas a seguir:

(1) **Leitura dos arquivos padrão:** O código lê os arquivos padrão já apresentados para servir como base na geração dos arquivos, permitindo a personalização do código gerado e facilitando a manutenção e reutilização do código.
(2) **Inicialização de variáveis e estruturas** Variáveis *StringBuilder* são inicializadas para construir as partes específicas do código final de forma eficiente, evitando criação excessiva de objetos *string* intermediários.
(3) **Geração de códigos para eventos:**
   - **Tradução dos Eventos:** Uma lista `listOfEvents` é criada para armazenar eventos de todos os autômatos. Um dicionário `eventMap` mapeia eventos a sequências de caracteres para acesso rápido. Macros são criadas para representar a posição do evento e simplificar comparações.
   - **Geração dos códigos para eventos não controláveis:** Funções para gerenciar eventos não controláveis são criadas e reservadas para edição do usuário final. O código verifica se o evento não é controlável e adiciona o nome do evento ao texto de código correspondente no *StringBuilder*.

A tradução dos eventos pode ser detalhada no algoritmo 1, considerando E uma lista vazia de eventos, $\Sigma(G_i)$ o conjunto de eventos do autômato G_i, e $G = G_{planta} \cup G_{supervisor}$ sendo o conjunto de autômatos do sistema. Considere $F_S(e)$ a função que codifica eventos em sequências binárias únicas, Σ_u o conjunto de eventos não controláveis do sistema, e $D_{e\leftrightarrow s}$ um dicionário que associa eventos a sequências de caracteres.

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

Entrada: Conjunto de autômatos G
Saída: Dicionário de eventos e sequências $D_{e\leftrightarrow s}$
início
  $E \leftarrow \varnothing$
  para *cada* G_i *em* G **faça**
    $E \leftarrow E \cup \Sigma(G_i)$
  $D_{e\leftrightarrow s} \leftarrow \varnothing$
  para *cada e em E* **faça**
    $s \leftarrow F_S(e)$
    $D_{e\leftrightarrow s}[e] \leftarrow s$
    se $e \in \Sigma_u$ **então**
      Cria a etapa da função de gerenciamento de eventos não controláveis referente a esse evento

(4) **Geração do código para todos os autômatos da lista:** Os autômatos são divididos entre aqueles que modelam a planta e os supervisores do sistema. Para cada autômato, é obtido o número de estados e são construídas as funções e as lógicas de transição para cada estado. Um dicionário é criado para mapear estados para valores inteiros, e as transições e eventos correspondentes são obtidos e armazenados. Um objeto é criado para cada autômato, contendo as informações e funções necessárias. Conforme destrinchado no algorítimo 2.
(5) **Substituição de texto nos arquivos de saída:** O código substitui *placeholders* nos arquivos padrão pelos blocos de código gerados para cada autômato, permitindo a integração do código gerado aos arquivos existentes.
(6) **Escrita dos arquivos de saída** Esta etapa escreve os resultados da conversão nos arquivos correspondentes.

Considerando $Q(a)$ o conjunto de estados de um automato e $len(S)$ como a função que retorna o número de estados da lista S, e $\Gamma(s)$ o conjunto dos eventos que podem ocorrer no estado s, temos V_e como o vetor de eventos fisicamente possíveis por estado.

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

Entrada:
G_s: Lista dos autômatos supervisores;
G_a: Lista dos autômatos plantas;
início
  $G = G_a \cup G_s$
  para *cada* G_i *em* G **faça**
    $S = Q(G_i)$, $n = len(S)$
    para *cada s de S* **faça**
      se $a \in G_a$ **então**
        Cria a definição e implementação da ação referente ao estado e adiciona ao vetor de ações de estado
      $V_e \leftarrow \Gamma(s)$
    se $a \in G_a$ **então**
      Cria a implementação da função de *loop* invocando o vetor de ações de estado
    senão
      Cria a implementação da função de *loop* invocando o vetor de posição única com uma função vazia
    Mapeia os estados para números inteiros
    Produz a implementação da função de transição usando os estados e eventos mapeados

3.4 Arquitetura da Rotina Principal

A rotina principal descreve a sequência de ações executadas pelo sistema durante o seu funcionamento normal. O algorítimo 3 detalha como a aplicação se comporta em execução continua no sistema embarcado.

Considerando que A representa o conjunto de Autômatos, Σ_u o conjunto dos eventos não controláveis, Σ_c dos eventos contro-

láveis, $E(t)$ a Função que retorna os eventos que ocorreram no instante t, $F_h(E)$ a função que retorna os eventos habilitados, $F_f(E)$ a função que retorna os eventos fisicamente possíveis, $F_{G_i}(s, G_i)$ a função que executa a transição para um dado evento e no autômato G_i, e $\delta(e, G_i)$ a função que executa a ação do estado atual s do autômato G_i.

Algoritmo 3 `mainDefault.INO`: Algoritmo da rotina principal

```
Σu, Σc ← E(t)  Σu ← Ff(Σu)  Σc ← Ff(Σc)
se ∃ Σu então
    para cada e ∈ Σu faça
        se e = 1 então
            para cada Gi ∈ G faça
                δ(e, Gi)
            interrompe
senão
    Σc ← Fh(Σc)
    se ∃ Σc então
        para cada e ∈ Σc faça
            se e = 1 então
                para cada Gi ∈ G faça
                    δ(e, Gi)
                interrompe
para cada Gi ∈ G faça
    FGi(s, Gi)
```

O ciclo principal da rotina é repetido continuamente, permitindo que os autômatos processem eventos, realizem transições de estado e executem suas lógicas específicas. As principais etapas dessa arquitetura são:

(1) **Obtenção dos eventos do sistema:** Obtém os eventos do sistema, divididos em eventos controláveis, eventos não controláveis.
(2) **Verificação de eventos factíveis:** Realiza uma operação de bit a bit entre os eventos que são fisicamente factíveis em todas as plantas e os eventos controláveis e não controláveis, identificando quais eventos podem ocorrer no sistema naquele momento.
(3) **Execução de transição de estados para eventos não controláveis:** Se houver eventos factíveis não controláveis, verifica-se o primeiro evento dessa categoria e executa a transição de estados.
(4) **Verificação de eventos habilitados:** Realiza uma operação de bit a bit entre os eventos que são estão habilitados em todas os supervisores e os eventos controláveis.
(5) **Execução de transição de estados para eventos controláveis:** Se não ocorreu a transição dos eventos não controláveis, verifica se existem eventos habilitados controláveis e executa a transição de estado em cada autômato.
(6) **Execução do *loop* do autômato:** Executa a função *loop* de cada autômato, que executa a ação referente ao estado atual.

4. ESTUDO DE CASO

Nesta seção, um estudo de caso para demonstrar a eficiência da solução desenvolvida será apresentado. Apresentando desde a modelagem dos SEDs utilizando o UltraDES até o resultado final com as alterações relacionadas à implementação de baixo nível.

O sistema a ser trabalhado é a Pequena Fábrica, conhecida como *Small Factory* (Wonham and Cai, 2017), uma planta muito relevante na área de Sistemas a Eventos Discretos. Ela consiste em duas máquinas conectadas por um *buffer* unitário, onde a primeira máquina trabalha no produto e o deposita no *buffer*, e a segunda máquina retira o produto do *buffer* para realizar outro trabalho sobre ele. A especificação de segurança da Pequena Fábrica visa evitar o *underflow* (quando a segunda máquina tenta retirar um produto do *buffer* vazio) e o *overflow* (quando a primeira máquina tenta colocar um produto no *buffer* cheio).

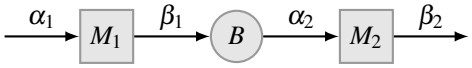
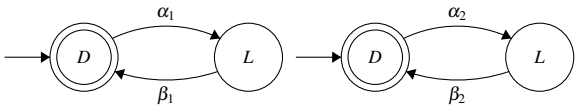


Figura 3. Diagrama da Pequena Fábrica

Modelagem usando SED: Definindo os eventos α_1 e α_2 como os que iniciam as máquinas $M1$ e $M2$, enquanto β_1 e β_2 representam o término da tarefa das maquinas, tem-se α_i como eventos controláveis e β_i como eventos não controláveis. Com base nessas informações, pode-se encontrar um modelo para essa planta com base nos autômatos para as máquinas 1 e 2, contendo dois estados: o estado inicial, marcado como o estado desligado, e um outro estado para a máquina ligada. A transição do estado desligado para o ligado ocorre pelos eventos α_i, e do estado ligado para o estado desligado é dado pelos eventos β_i.



(a) M_1 (b) M_2

Figura 4. Modelo das plantas para a pequena fábrica

Para M_1, uma tarefa completa é depositar uma peça no *buffer*; para M_2, uma tarefa completa é retirar uma peça do *buffer* e trabalhar na mesma até entregá-la pronta

A especificação de segurança representa a restrição imposta pela presença do *buffer* e pode ser modelada de forma que exista um estado que representa o *buffer* vazio, enquanto outro estado representa o *buffer* cheio. O *buffer* fica cheio quando a máquina 1 termina seu processo (evento β_1) e fica vazio quando a máquina 2 começa seu processamento (evento α_2).

Algoritmo 4 Exemplo UltraDES : Pequena fabrica

```
1  // Automato Maquina 1 e 2
2  var G1 = new DeterministicFiniteAutomaton(new[]{
3      new Transition(S11, a1, S12),
4      new Transition(S12, b1, S11)
5    }, S11, "G1");
6  var G2 = new DeterministicFiniteAutomaton(new[]{
7      new Transition(S21, a2, SB2),
8      new Transition(SB2, b2, S21)
9    }, S21, "G2");
10 // Automato Buffer (Especificacao)
11 var E = new DeterministicFiniteAutomaton(new[]{
12     new Transition(SB1, b1, SB2),
13     new Transition(SB2, a2, SB1)
14   }, SB1, "E");
15 var Supervisor = DeterministicFiniteAutomaton.
       MonolithicSupervisor(new[]{ G1, G2 },new[]{E},true);
```

Implementação no UltraDES: Com base nessa analise e utilizando o UltraDES modelou-se esse sistema da forma descrita no algorítimo 4. Utilizou-se também o UltraDES para chegar ao supervisor do sistema utilizando da abordagem monolítica

que irá garantir que o funcionamento do sistema siga a especificação.

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, também é criado uma lista para todos os supervisores. Por fim, é chamada a função `ConvertDEStoINO`. Para exemplificar como se deu a conversão, segue o caso da máquina 1:

Populando a lista de autômatos

```
Automaton automata[NUMBER_AUTOMATON];
Automaton supervisor[NUMBER_SUPERVISOR];
automata[0] = Automaton(2,enabledEventStatesAutomaton0,&
      MakeTransitionAutomaton0,&Automaton0Loop);
```

Aqui são instanciadas as listas de autômatos referentes às plantas e aos supervisores. Para cada elemento dessas listas, é instanciado um autômato com base no número de estados, lista de eventos habilitados por estado, função de transição e função das ações.

Representação da transição

```
int MakeTransitionAutomaton0(int State,Event event){
if(State == 0 && (getBit(event,EVENT_A1))) return 1;
if(State == 1 && (getBit(event,EVENT_B1))) return 0;
return(State);}
```

Um exemplo da implementação da função de transição no caso para o automato referente a máquina 1, nela percebe-se que se a dinâmica vista na figura 4.

Representação dos Estados: *Função de Ação do Estado:* A função de ação do estado D_1 da máquina M_1

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

Essa função chama um vetor de funções em que o índice representa o estado em que o automato se encontra no isentante atual. *Eventos Habilitados do Estado:* Os eventos habilitados do estado D_1 são representados pelo vetor de eventos habilitados:

```
uint8_t eventDataAUTOEV0[1] = {0b11110111};
uint8_t eventDataAUTOEV1[1] = {0b11111011};
Event enabledEventStatesAutomaton0[2] = { createEventFromData(
      eventDataAUTOEV0), createEventFromData(eventDataAUTOEV1)};
```

Número do Estado: No exemplo, o estado D_1 da máquina M_1 é identificado pelo número 0 na função de ação do estado.

Implementação de Baixo Nível: Após a conversão é necessário faz alterações no arquivo `UserFunctions.cpp` para garantir o funcionamento correto do sistema.

Configuração dos pinos

```
#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);}
```

Aqui são configurados os pinos de entrada e saída.

Implementação das ações de estados

```
void StateAction10(){digitalWrite(OUTM1,LOW); }
void StateAction11(){digitalWrite(OUTM1,HIGH);}
void StateAction20(){digitalWrite(OUTM2,LOW); }
void StateAction21(){digitalWrite(OUTM2,HIGH);}
```

As funções `StateAction10` e `StateAction20` desligam as máquinas 1 e 2. Já as funções `StateAction11` e `StateAction21` ligam as máquinas 1 e 2. Essas ações são realizadas usando a instrução `digitalWrite`.

4.1 Resultados

Após a montagem física da planta e o *upload* do programa para o *NodeMCU* pode-se observar um desempenho consistente e sem falhas, confirmando a eficácia do algoritmo implementado.

5. CONCLUSÃO

O trabalho [1] oferece uma solução que automatiza a implementação de sistemas de eventos discretos ao converter autômatos finitos, modelados pela biblioteca UltraDES, em código compatível com o Arduino. A abordagem proposta é prática e eficaz, facilitando a tradução da lógica de automação e tornando-a acessível para projetos acadêmicos.

Além disso, o projeto demonstra a viabilidade e utilidade dessa abordagem, oferecendo uma maneira mais acessível e intuitiva de desenvolver soluções de automação baseadas em SEDs. A disponibilização da biblioteca UltraDES e a documentação detalhada da metodologia empregada fornecem uma ferramenta poderosa para a comunidade acadêmica, impulsionando o avanço das pesquisas em eventos discretos. Isso permite que os pesquisadores se concentrem em aspectos mais específicos de seus projetos, sem se preocupar com a implementação detalhada da lógica de automação.

REFERÊNCIAS

Alves, L., Martins, L., and Pena, P. (2017). Ultrades - a library for modeling, analysis and control of discrete event systems. *IFAC-PapersOnLine*, 50, 5831–5836. doi: 10.1016/j.ifacol.2017.08.540.

Cassandras, C. and Lafortune, S. (2010). *Introduction to Discrete Event Systems*. doi:10.1007/978-0-387-68612-7.

Fabian, M. and Hellgren, A. (1998). Plc-based implementation of supervisory control for discrete event systems. volume 3, 3305 – 3310 vol.3. doi:10.1109/CDC.1998.758209.

Hering de Queiroz, M. and Cury, J. (2000). Modular control of composed systems. volume 6, 4051 – 4055 vol.6. doi: 10.1109/ACC.2000.876983.

Hering de Queiroz, M. and Cury, J. (2002). Synthesis and implementation of local modular supervisory control for a manufacturing cell. 377 – 382. doi: 10.1109/WODES.2002.1167714.

Wonham, W. and Ramadge, P. (1988). Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems*, 1(1), 13–30. doi: 10.1007/BF02551233.

Wonham, W. and Cai, K. (2017). *Supervisory Control of Discrete-Event Systems, v.20170901*.

[1] Para acessar o código completo e funcional, por favor visite o repositório GitHub do projeto: https://github.com/lacsed/arduino-generator.git