

# **Geração de código para máquina de estados em sistemas embebidos**

**José Pedro Magalhães Marrafa**



**FEUP** FACULDADE DE ENGENHARIA  
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Carlos João Rodrigues Costa Ramos

24 de junho de 2019

© José Marrafa, 2019

# Resumo

A presente dissertação teve como objetivo o desenvolvimento de uma *interface* que permitisse facilitar a implementação de um sistema de controlo de eventos discretos num sistema embebido.

O sistema de controlo assume a forma de máquina de estados e o utilizador pode utilizar uma *desktop application*, desenvolvida para *Windows* que permite a especificação tabelar do comportamento de uma ou múltiplas máquinas, utilizando o conceito de ortogonalidade.

As funcionalidades que lhes podem ser associadas passam por uma abordagem híbrida dos formalismos de modelação das máquinas de *Moore* e *Mealy* e da norma *UML* para *statecharts*, permitindo a existência de ações associadas aos estados que ocorrem à sua entrada, saída ou continuamente, bem como ações associadas a eventos que são executadas no disparo de uma transição.

Uma das funcionalidades diferenciadoras em relação aos *softwares* semelhantes que já existem no mercado, consiste na possibilidade de associar funções às ações que alterem o comportamento de outras máquinas de estado, baseadas nas macro-ações da norma *GRAFCET*, permitem efetuar o congelamento, descongelamento, desativação ou forçagem de um estado de máquinas que se encontram noutras regiões ortogonais.

Tendo como propósito a sua implementação num sistema embebido, a *UI (User Interface)* permite a escolha entre uma geração de código *C* genérico que pode ser implementado em micro-controladores/microprocessadores que suportem esta linguagem de programação, ou em *Arduino (C++)*, permitindo associar às ações e transições da máquina funções como a ativação ou a desativação de pinos, e condições baseadas nos valores dos pinos de entrada, nos estados ativos de outras máquinas ou em eventos temporais. Para além disso o utilizador pode escolher qual o método de modelação da máquina pretendido, tendo à sua disposição as soluções mais populares de codificação de máquinas de estados.

A sua utilização permite a obtenção rápida de código que, caso fosse produzido manualmente, conteria instruções repetitivas, extensas e propícias a darem origem a erros. Pode ser utilizada por programadores experientes em codificação embebida, de forma a reduzir significativamente o tempo de conversão deste tipo de modelos gráficos em código, mas também por programadores com menos experiência que pretendam implementar, de forma direta, num sistema embebido, os grafismos que projetaram, sem ter necessariamente de se familiarizar com todos os métodos de implementação de máquinas de estado em *C*.



# Abstract

The goal of this dissertation was to develop an interface that simplified the implementation of a discrete event control system in an embedded system.

The control system acts like a state machine and the user uses a desktop application, developed for Windows which allows the specification of the behavior of one or more state machines operating in parallel specified on several tables.

The functionalities they're known for are a hybrid approach of the formalisms to model Moore and Mealy machines, as well as the use of the UML statecharts notation, allowing the existence of actions associated with states occurring at their entry, exit or continuously, as well as actions associated with events that are executed in the firing of a transition.

One of the innovative functionalities when compared to similar software that already exists in the market is the possibility of associating to these actions functionalities that change the behavior of other existing state machines in the project, based on the macro-actions of the GRAFCET standart, which allow the freezing, unfreezing, deactivation or enforce the state of other machines that exist in other orthogonal regions.

The goal is to implement the code in an embedded system, so the UI (User Interface) allows the choice between the generation of C code which can be implemented in microcontrollers/ microprocessors that support this language, or Arduino (C++) allowing to associate to the actions and transitions of the machine functions as the activation or deactivation of pins, conditions based on the values of the input pins, based on the active states of other machines or based on temporal events. In addiction, the user can choose which method of modelling the machine he wants, having at his disposal the most popular state machine coding solutions.

With its help, code generating is swift and without repetitive and widespread errors that would likely occur if it were typed manually. It can be used by programmers experienced in embedded coding to cut-down the time it takes to convert these graphic models into code, but also by less experienced programmers who want to implement the graphic models they designed straight into an embedded system, without necessarily having to know every implementation method of state machines in C.



# Agradecimentos

Gostaria de começar por agradecer aos meus pais, que sempre me instruíram a mentalidade de seguir com os meus estudos, que só terminariam com uma formação superior, e por todo apoio e motivação que me deram.

À minha família, com ênfase nos meus avós que sempre demonstraram interesse e preocupação pela conclusão dos meus estudos, bem como por todo o suporte financeiro que disponibilizaram de forma a que tal fosse possível.

Ao meu orientador, o Professor Doutor Carlos João Ramos, por toda a disponibilidade que demonstrou ao longo do semestre e pela forma objetiva e paciente com que me guiou, permitindo que o tempo despendido para desenvolvimento deste projeto passasse sempre pela execução de tarefas que trouxeram valor para a solução final.

Ao Rafael Gonçalves, por ter utilizado a aplicação desenvolvida no âmbito desta dissertação para gerar código para a sua tese, o que consistiu num importante componente de validação da ferramenta que foi desenvolvida.

À minha namorada, aos meus colegas e amigos que me acompanharam neste percurso, com uma menção especial para o Diogo, Henrique, João, José, Luís, Rui e Sebastião, obrigado por todo o companheirismo e disponibilidade para ajudar nos momentos mais difíceis.

Um grande obrigado a todos,

José Marrafa



*“Be the change that you wish to see in the world.”*

Mahatma Gandhi



# Conteúdo

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introdução</b>   | <b>1</b>  |
| 1.1      | Enquadramento . . . . .   | 1         |
| 1.2      | Motivação . . . . .   | 2         |
| 1.3      | Objetivos . . . . .   | 2         |
| 1.4      | Estrutura . . . . .   | 3         |
| <b>2</b> | <b>Revisão Bibliográfica</b>  | <b>5</b>  |
| 2.1      | Máquinas de estados . . . . .   | 5         |
| 2.1.1    | Modelação tradicional . . . . .   | 6         |
| 2.1.2    | Modelação tabelar . . . . .   | 7         |
| 2.1.3    | Modelação hierárquica . . . . .   | 7         |
| 2.2      | Sistemas Embebidos . . . . .  | 10        |
| 2.3      | Linguagens utilizadas para programação embebida . . . . .                     | 11        |
| 2.4      | Métodos de estruturação de uma máquina de estados em C . . . . .              | 12        |
| 2.4.1    | Expressões condicionais . . . . .   | 13        |
| 2.4.2    | Apontadores para funções . . . . .  | 15        |
| 2.5      | Softwares semelhantes . . . . .   | 17        |
| 2.5.1    | <i>QM<sup>TM</sup> modelling tool</i> . . . . .                               | 17        |
| 2.5.2    | <i>Simulink</i> . . . . .   | 18        |
| 2.5.3    | <i>IAR Visual State</i> . . . . .   | 18        |
| 2.5.4    | <i>YAKINDU Statechart Tools</i> . . . . .                                     | 19        |
| 2.5.5    | Outros . . . . .  | 19        |
| <b>3</b> | <b>Desenvolvimento</b>  | <b>21</b> |
| 3.1      | Tecnologias . . . . .   | 21        |
| 3.1.1    | Plataformas de desenvolvimento de uma <i>UI</i> para <i>Windows</i> . . . . . | 22        |
| 3.1.2    | Escolha da plataforma e do <i>IDE</i> . . . . .                               | 22        |
| 3.1.3    | Geração de código utilizando <i>T4 Text Templates</i> . . . . .               | 24        |
| 3.2      | Funcionalidades das máquinas de estados criadas na aplicação . . . . .        | 26        |
| 3.3      | Funcionalidades do código gerado pela aplicação . . . . .                     | 27        |
| 3.4      | <i>User Interface</i> . . . . .   | 28        |
| 3.4.1    | Manual de Utilização . . . . .  | 29        |
| 3.5      | Código gerado . . . . .   | 37        |
| 3.5.1    | Ações hierárquicas . . . . .  | 42        |
| <b>4</b> | <b>Testes e Resultados</b>  | <b>45</b> |
| 4.1      | Problema 1 - Controlo de dois semáforos . . . . .                             | 45        |
| 4.1.1    | Descrição do Problema . . . . .   | 45        |

|          |  |           |
|----------|--|-----------|
| 4.1.2    | Modelação na <i>interface</i> da aplicação . . . . .       | 47        |
| 4.1.3    | Resultados da implementação do código gerado . . . . .     | 53        |
| 4.2      | Problema 2 - Controlo de uma <i>UPS</i> portátil . . . . . | 55        |
| 4.2.1    | Descrição do Problema . . . . .                            | 55        |
| 4.2.2    | Modelação na <i>interface</i> da aplicação . . . . .       | 57        |
| 4.2.3    | Resultados da implementação do código gerado . . . . .     | 63        |
| <b>5</b> | <b>Conclusões e Trabalho Futuro</b>                        | <b>65</b> |
| 5.1      | Cumprimento dos objetivos . . . . .                        | 65        |
| 5.2      | Trabalho Futuro . . . . .                                  | 67        |
| <b>A</b> | <b>Anexos</b>  | <b>69</b> |
| A.1      | Código gerado pela aplicação para o Problema 1 . . . . .   | 69        |
|          | <b>Referências</b>   | <b>85</b> |

# **Lista de Figuras**

|      |  |    |
|------|--|----|
| 2.1  | Modelação de uma máquina de <i>Moore</i> . . . . .   | 6  |
| 2.2  | Modelação de uma máquina de <i>Mealy</i> . . . . .   | 6  |
| 2.3  | Representação gráfica de estados, transições, eventos e ações em <i>UML</i> . . . . .          | 8  |
| 2.4  | Representação gráfica de um estado e suas funcionalidades em <i>UML</i> . . . . .              | 9  |
| 2.5  | Representação gráfica de uma transição e suas funcionalidades em <i>UML</i> . . . . .          | 9  |
| 2.6  | Representação gráfica de regiões ortogonais em <i>UML</i> . . . . .                            | 10 |
| 2.7  | Modelação de uma máquina de estados genérica em UML . . . . .                                  | 13 |
| 2.8  | Implementação de uma máquina de estados, em C, utilizando <i>switch-case</i> . . . . .         | 14 |
| 2.9  | Implementação de uma máquina de estados, em C, utilizando <i>if-else</i> . . . . .             | 14 |
| 2.10 | Implementação, em C, de uma abordagem utilizando apontadores para funções . . . . .            | 16 |
| 2.11 | <i>Interface</i> do software <i>QM<sup>TM</sup></i> . . . . .                                  | 17 |
| 2.12 | <i>Interface</i> da biblioteca <i>stateflow</i> em <i>Simulink</i> . . . . .                   | 18 |
| 2.13 | <i>Interface</i> do software <i>IAR Visual Studio</i> . . . . .                                | 18 |
| 2.14 | <i>Interface</i> da ferramenta <i>YAKINDU Statechart Tools</i> no <i>IDE Eclipse</i> . . . . . | 19 |
| 3.1  | Diagrama das diferentes fases de desenvolvimento do projeto . . . . .                          | 21 |
| 3.2  | <i>Design</i> do botão "Generate Code" . . . . .   | 23 |
| 3.3  | Instruções em "XAML" para gerar o botão . . . . .  | 23 |
| 3.4  | Método "GenerateCode Click" . . . . .  | 23 |
| 3.5  | <i>MessageBox</i> que surge após <i>click</i> do botão . . . . .                               | 23 |
| 3.6  | <i>T4 Text Template</i> sem blocos lógicos . . . . .   | 24 |
| 3.7  | <i>T4 Text Template</i> com blocos lógicos . . . . .   | 24 |
| 3.8  | Envio de dados para o <i>template</i> , utilizando C# . . . . .                                | 25 |
| 3.9  | Diagrama das funcionalidades associadas ao código gerado na aplicação . . . . .                | 27 |
| 3.10 | Desenvolvimento da camada <i>front-end</i> da aplicação no <i>WPF</i> . . . . .                | 28 |
| 3.11 | Desenvolvimento da camada <i>back-end</i> da aplicação no <i>WPF</i> . . . . .                 | 28 |
| 3.12 | <i>Interface</i> de escolha da linguagem dos ficheiros gerados . . . . .                       | 29 |
| 3.13 | <i>Interface</i> da aplicação no caso da escolha <i>Arduino</i> . . . . .                      | 29 |
| 3.14 | Tabela de definição de pinos . . . . .   | 30 |
| 3.15 | Tabela de definição dos estados e das suas ações . . . . .                                     | 30 |
| 3.16 | Associação de um evento temporal a uma transição . . . . .                                     | 31 |
| 3.17 | Tabela de definição das transições e das suas ações . . . . .                                  | 31 |
| 3.18 | Conjunto de funcionalidades que podem ser associadas às ações . . . . .                        | 32 |
| 3.19 | <i>Interface</i> que permite a introdução de instruções manualmente . . . . .                  | 32 |
| 3.20 | Conjunto de funcionalidades que podem ser associadas às transições . . . . .                   | 33 |
| 3.21 | <i>Interface</i> da secção para geração do ficheiro de código . . . . .                        | 33 |
| 3.22 | Ícones de informação . . . . .   | 34 |
| 3.23 | Exemplo de <i>MessageBox</i> de erro no não cumprimento de normas de modelação .               | 34 |

|  |    |
|--|----|
| 3.24 <i>Interface</i> da abertura de um <i>OpenFileDialog</i> . . . . .                                  | 35 |
| 3.25 Ficheiro <i>XML</i> de um projeto gravado na aplicação . . . . .                                    | 35 |
| 3.26 Atribuição manual de um nome a um estado . . . . .  | 36 |
| 3.27 Atribuição automática de um nome a um estado . . . . .  | 36 |
| 3.28 Atribuição automática de um nome a um estado . . . . .  | 36 |
| 3.29 Secção de código para geração de um <i>switch</i> para cada máquina, utilizando <i>T4</i> . . . . . | 38 |
| 3.30 Código gerado como consequência das instruções da figura anterior . . . . .                         | 38 |
| 3.31 Secção de código para geração de um <i>switch</i> para cada máquina, utilizando <i>T4</i> . . . . . | 38 |
| 3.32 Código gerado como consequência das instruções da figura anterior . . . . .                         | 39 |
| 3.33 Secção de código para geração de ações à entrada de um estado, utilizando <i>T4</i> . . . . .       | 39 |
| 3.34 Código gerado como consequência das instruções da figura anterior . . . . .                         | 40 |
| 3.35 Secção de código para geração de condições de transição, utilizando <i>T4</i> . . . . .             | 40 |
| 3.36 Código gerado como consequência das instruções da figura anterior . . . . .                         | 40 |
| 3.37 Ficheiros gerados na escolha da linguagem <i>C</i> . . . . .  | 41 |
| 3.38 Inclusão dos ficheiros gerados no projeto do utilizador . . . . .                                   | 41 |
| 3.39 Exemplo de uma implementação para o microcontrolador TMS320F2803x . . . . .                         | 41 |
| 3.40 Variáveis <i>booleanas</i> geradas para execução de ações hierárquicas . . . . .                    | 42 |
| 3.41 Código gerado para execução de ações de congelamento . . . . .                                      | 42 |
| 3.42 Alterações no código da máquina alvo de uma ação de congelamento . . . . .                          | 43 |
| 3.43 Alterações no código da máquina alvo de uma ação de desativação . . . . .                           | 43 |
| 3.44 Alterações no código da máquina alvo de uma ação de desativação . . . . .                           | 44 |
| 3.45 Código gerado para execução de ações de forçagem de estado . . . . .                                | 44 |
| 3.46 Alterações no código da máquina alvo de uma ação de forçagem de estado . . . . .                    | 44 |
| <br>   |    |
| 4.1 <i>UML statechart</i> para controlo de 2 semáforos . . . . .   | 46 |
| 4.2 Diagrama das fases necessárias para obtenção e implementação do código . . . . .                     | 47 |
| 4.3 <i>Interface</i> de escolha da linguagem dos ficheiros gerados . . . . .                             | 48 |
| 4.4 <i>Interface</i> da aplicação com um projeto vazio . . . . .   | 48 |
| 4.5 Tabela " <i>Pin Definition</i> " com os parâmetros do problema 1 . . . . .                           | 49 |
| 4.6 Tabela " <i>States</i> " com os parâmetros da primeira máquina . . . . .                             | 49 |
| 4.7 Tabela " <i>Transitions</i> " com os parâmetros da primeira máquina . . . . .                        | 50 |
| 4.8 Tabelas " <i>States</i> " com os parâmetros da segunda, terceira e quarta máquina . . . . .          | 51 |
| 4.9 Tabelas " <i>Transitions</i> " com os parâmetros da segunda, terceira e quarta máquina . . . . .     | 51 |
| 4.10 <i>Interface</i> da secção para geração do ficheiro de código . . . . .                             | 52 |
| 4.11 Instruções de código geradas por consequência da tabela " <i>Pin Definition</i> " . . . . .         | 52 |
| 4.12 Introdução dos endereços dos pinos utilizados . . . . .   | 52 |
| 4.13 Inicialização das variáveis utilizadas na função " <i>Code( )</i> " . . . . .                       | 53 |
| 4.14 Circuito montado para implementação do problema 1 . . . . .   | 53 |
| 4.15 Impressão, via porta série, dos estados ativos . . . . .  | 54 |
| 4.16 Impressão, via porta série, dos estados ativos na ocorrência de ações hierárquicas                  | 54 |
| 4.17 <i>UML statechart</i> para controlo de uma <i>UPS</i> portátil . . . . .                            | 56 |
| 4.18 Diagrama das fases necessárias para obtenção e implementação do código . . . . .                    | 57 |
| 4.19 <i>Interface</i> de escolha da linguagem dos ficheiros gerados . . . . .                            | 57 |
| 4.20 Tabela <i>Pin definition</i> com os parâmetros do problema 2 . . . . .                              | 57 |
| 4.21 Associação de funções com um único parâmetro a uma ação de saída . . . . .                          | 58 |
| 4.22 Associação de funções com múltiplos parâmetros a uma ação de entrada . . . . .                      | 58 |
| 4.23 Associação de funções com múltiplos parâmetros a múltiplas ações de entrada . . . . .               | 59 |
| 4.24 Associação de funções com múltiplos parâmetros a múltiplas ações de entrada . . . . .               | 59 |
| 4.25 Associação de eventos com um único parâmetro a uma transição . . . . .                              | 60 |

|   |    |
|---|----|
| 4.26 Associação de eventos com um único parâmetro a uma transição . . . . .           | 60 |
| 4.27 Associação de eventos com múltiplos parâmetros a múltiplas transições . . . . .  | 61 |
| 4.28 Associação de eventos com múltiplos parâmetros a múltiplas transições . . . . .  | 61 |
| 4.29 Geração do código para o problema 2 . . . . .                                    | 61 |
| 4.30 Introdução manual dos endereços dos pinos do <i>Arduino</i> utilizados . . . . . | 62 |
| 4.31 Abertura do projeto criado . . . . .   | 62 |
| 4.32 Montagem da <i>UPS</i> portátil . . . . .  | 63 |
| 4.33 Resultados obtidos . . . . .   | 63 |



# Abreviaturas e Símbolos

|         |   |
|---------|---|
| ASIC    | Application Specific Integrated Circuits            |
| CAN     | Controller Area Network                             |
| CSS     | Cascading Style Sheets                              |
| EEPROM  | Electrically Erasable Programmable Read-Only Memory |
| FPGA    | Field Programmable Gate Array                       |
| GRAFCET | GRAphe Fonctionnel de Commande, Étapes Transitions  |
| HDL     | Hardware Description Language                       |
| HTML    | HyperText Markup Language                           |
| I2C     | Inter-Integrated Circuit                            |
| IDE     | Integrated Development Environment                  |
| LED     | Light Emitting Diode                                |
| MOSFET  | Metal Oxide Semiconductor Field Effect Transistor   |
| NASA    | National Aeronautics and Space Administration       |
| OS      | Operating System                                    |
| PAC     | Programmable Automation Controller                  |
| PC      | Personal Computer                                   |
| PLC     | Programmable Logic Controller                       |
| RAM     | Random-Access Memory                                |
| ROM     | Read-Only Memory                                    |
| RTC     | Run to Completion                                   |
| SCXML   | State Chart XML                                     |
| SM      | State Machine                                       |
| SPI     | Serial Peripheral Interface                         |
| ST      | Structured Text                                     |
| UART    | Universal Asynchronous Receiver-Transmitter         |
| UI      | User Interface                                      |
| UML     | Unified Modeling Language                           |
| UPS     | Uninterruptible Power Supply                        |
| USB     | Universal Serial Bus                                |
| UWP     | Universal Windows Platform                          |
| VHDL    | VHSIC Hardware Description Language                 |
| VHSIC   | Very High Speed Integrated Circuits                 |
| WPF     | Windows Presentation Foundation                     |
| XML     | eXtended Markup Language                            |
| XAML    | eXtended Application Markup Language                |



# Capítulo 1

## Introdução

### 1.1 Enquadramento

O mundo à nossa volta encontra-se em constante avanço científico e, à medida que o horizonte tecnológico expande, torna-se cada vez mais evidente que tudo o que nos rodeia é tornado inteligente e automatizado. Muita dessa automatização é realizada através da programação de unidades de processamento, utilizando sistemas embebidos, verificando-se, cada vez mais, um acréscimo da sua importância. [1]

Este tipo de sistemas encontra-se presente na grande generalidade dos dispositivos utilizados pela sociedade, tais como: máquinas de venda de produtos, nos quais efetuam a libertação dos produtos quando o valor correto de moedas é introduzido, em elevadores, onde sequenciam as paragens deste mesmo de acordo com o pedido do utilizador, em semáforos, coordenando a sua sequência de fases, bem como em telemóveis, em aplicações domésticas (televisões, micro-ondas, portões de garagem, máquinas de lavar), ou mesmo em automóveis (sistemas de travagem, ativação do *airbags*, aplicações de *cruise control*).

Um sistema embebido é uma aplicação que contém pelo menos um computador programável (tipicamente sobre a forma de microcontrolador, microprocessador ou *chip* de processamento digital de sinal) e que é utilizado por indivíduos que, geralmente, desconhecem que esse sistema é computado. [2]

São sistemas de controlo de aplicações que se encontram integrados no próprio dispositivo ou sistema que controlam, interagindo com o mundo exterior através de sensores e atuadores.

A sua função é a de realizar um conjunto de tarefas predefinidas, facilitando a execução de tarefas de automação, de controlo, ou de medição precisa, de forma simples e eficaz, [3] contrastando, por isso, com a funcionalidade de um *PC* convencional que é desenhado para realizar múltiplas tarefas.

As linguagens de programação mais utilizadas neste tipo de sistema são o *C* e o *C++*. No entanto, existe uma grande variedade de linguagens que podem ser utilizadas tais como *Python*, *Java*, *Rust*, *Ada*, *JavaScript*, *Go*, *Lua*, *Assembly*, *Verilog* ou *VHDL*.

## 1.2 Motivação

A escolha das ferramentas de modelação apropriadas é fundamental para a resolução de problemas que envolvem sistemas automatizados complexos e que apresentam um grande número de requisitos. [4]

As ferramentas de modelação como as máquinas de estados permitem modelar controladores de eventos discretos, facilitando, portanto, a sua implementação.

A utilização de máquinas de estados é um dos padrões mais populares para desenho de aplicações em sistemas embebidos, mas o facto de estes modelos serem expressos graficamente torna a sua aplicação direta difícil. Surge, por isso, a necessidade de recorrer a técnicas que transformem os modelos de controlo em código que possa ser implementado neste tipo de sistemas.

Estas técnicas têm de ser codificadas manualmente e consistem, geralmente, em sequências repetitivas de instruções condicionais como o "*switch-case*" ou o "*if-else*" ou em apontadores para funções. À medida que o número de estados que constituem a máquina aumenta, cresce também a probabilidade do surgimento de erros, já que o esquecimento de, por exemplo, uma simples instrução "*break*" compromete o bom funcionamento de todo o sistema.

A existência de ferramentas que permitam a implementação direta dessas técnicas, possibilitando a modelação da máquina de estados numa *UI* e a obtenção instantânea do seu código correspondente, facilitam a aplicação destes modelos em *software* embebido.

Para além de darem a possibilidade a um programador experiente de reduzir significativamente o tempo de implementação de uma máquina de estados, proporcionam também a possibilidade de um utilizador com menos experiência em programação embebida, implementar o comportamento desejado do sistema sem ter de desenvolver o seu código.

## 1.3 Objetivos

O objetivo deste projeto visa a criação de uma ferramenta que permita modelar sistemas de controlo de eventos discretos sobre a forma de máquinas de estados e obter o seu código equivalente numa linguagem de programação embebida, possível de ser diretamente implementado num microprocessador/microcontrolador.

Para tal, devem ser definidas as funcionalidades que as máquinas de estados modeladas na aplicação podem efetuar, tendo como base os formalismos de modelação existentes como as máquinas de *Moore* e *Mealy*, bem como os *Harel* e *UML statecharts*.

O utilizador deve poder especificar preferências na forma de geração do código, como a escolha da linguagem, das técnicas de estruturação do código gerado (*switch-case*, *if-else*, *function pointers*), o nome dos ficheiros gerados, dos estados, de variáveis utilizadas no código gerado, assim como a possibilidade de introdução de instruções de código, ao seu critério, através a *interface* da aplicação.

De forma a ser competitiva com o que já existe no mercado, devem ser analisados *softwares* semelhantes à aplicação a desenvolver e atribuir-lhe algumas funcionalidades diferenciadoras,

uma das quais foi definida desde o início e passaria pela possibilidade de associar ações baseadas nas macro-ações da norma GRAFCET, que permitissem congelar, descongelar, desativar e forçar um estado de outras máquinas do projeto. Outro fator de valorização passa pela apresentação de uma *interface* simples e intuitiva, que permita ao utilizador uma obtenção rápida, fácil e correta do ficheiro de código que corresponda às máquinas de estados especificadas, bem como o impedimento de tentativas de introdução de comportamentos que não cumpram a norma de construção de uma máquina de estados e a inclusão de funcionalidades que permitam a reutilização da aplicação, como a gravação, abertura e edição de projetos nela criados.

## 1.4 Estrutura

Esta dissertação está organizada em cinco capítulos:

- O presente capítulo contém uma contextualização do problema, seguida duma especificação dos objetivos e motivação do trabalho proposto.
- No capítulo 2 é efetuada uma revisão bibliográfica do problema a tratar, onde são abordados os formalismos de modelação de máquinas de estados, as principais linguagens utilizadas para programação em sistemas embebidos e realizada uma breve contextualização deste tipo de sistemas. São mencionadas também as formas de estruturação de uma máquina de estados em código e analisados softwares semelhantes ao desenvolvido no contexto desta dissertação.
- No capítulo 3 são mencionadas as tecnologias utilizadas para desenvolvimento do presente projeto, as funcionalidades que a aplicação e o código nela gerado permitem efetuar. Para além disso, é explicado o funcionamento da aplicação, com recurso a um breve manual de utilização e explicadas secções do código gerado.
- No capítulo 4 são delineados e montados problemas reais que utilizam máquinas de estados para modelação do seu comportamento, nos quais estão incluídos um sistema de controlo de semáforos e de uma UPS portátil. O código implementado nestes sistemas foi gerado utilizando apenas a aplicação desenvolvida, sendo apresentados os resultados obtidos e explicado, passo a passo, o processo de modelação e obtenção de código utilizando a *interface* da aplicação.
- No capítulo 5 são retiradas conclusões do projeto que foi desenvolvido e mencionadas as funcionalidades que podem ser implementadas num trabalho futuro de forma a melhorar o projeto desenvolvido.
- No anexo A pode-se encontrar o código que foi automaticamente gerado pela aplicação e que correspondente ao sistema de controlo de semáforos modelado no capítulo 4.



## Capítulo 2

# Revisão Bibliográfica

### 2.1 Máquinas de estados

Uma máquina de estados é um método de modelação de controladores orientados a sistemas de eventos discretos, denominados também de sistemas reativos devido ao seu comportamento onde a ocorrência de um evento implica que o sistema volte ao modo de espera da sua ocorrência seguinte.

Numa máquina de estados finita existe um número finito de estados que representam os diferentes comportamentos possíveis do sistema que se pretende controlar. Estes encontram-se ligados entre si através de transições, transitando para o estado seguinte na ocorrência de eventos. Quando o sistema se encontra num determinado estado, tal implica que o sistema apenas pode transitar para determinados estados, responder a um número limitado de eventos e produzir certas ações. Sendo assim, uma máquina de estados pode ser definida como parte de um sistema que se comporta de forma diferente, consoante eventos que ocorreram no passado. [5].

Quando um evento é processado, a máquina de estados responde efetuando ações, que podem consistir na mudança de uma variável, na atualização de entradas/saídas, na invocação de funções, na geração de outro evento ou na ocorrência de uma transição do estado atual do sistema para outro.

Estes modelos podem assumir formalismos mais tradicionais, como as máquinas de *Mealy* ou *Moore*, ou mais complexos como os *Harel* ou *UML statecharts*, em todos eles é assumido que a máquina de estados completa o processamento de cada evento, antes de iniciar o processamento do próximo, este modelo de execução é denominado de *run to completion*, ou *RTC*.

Consequentemente, a chegada de novos eventos não provoca a interrupção do evento que se encontra a ser processado, sendo estes guardados até que a máquina fique livre para os processar. Posto isto, o modelo de execução *RTC*, permite evitar o eventual surgimento de problemas provenientes do processamento de ações associadas às transições, visto que durante esse período o sistema não responde a eventos que possam ocorrer, algo que poderia ser problemático, já que a máquina de estados não se encontra em nenhum estado bem definido, uma vez que o estado correto do sistema seria o intermédio entre os dois estados.

## 2.1.1 Modelação tradicional

Segundo a *automata theory*, um estado é uma situação do sistema que depende de *inputs* anteriores e que causa uma reação nos *inputs* seguintes. Esta teoria usa os termos de *input/output* sobre a forma de texto simples, em contrapartida, máquinas de estado mais modernas usam uma definição mais extensa de *inputs/outputs*, podendo estes assumir a forma de eventos ou ações mais complexas.

Na modelação tradicional existem dois formalismos básicos para modelar uma máquina de estados finita: as máquinas de *Moore* e as máquinas de *Mealy*. Sendo que, é possível efetuar a transformação de um modelo no outro sem que as suas características sejam perdidas.

### 2.1.1.1 Máquinas de *Moore*

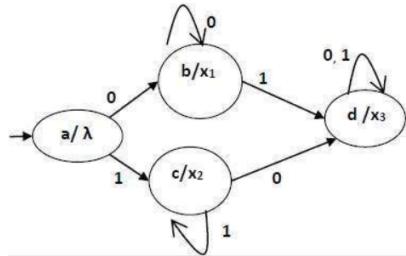


Figura 2.1: Modelação de uma máquina de *Moore* [6]

Nomeada em homenagem ao seu inventor *Edward Moore*, nas máquinas de *Moore* os estados são os responsáveis por produzir os *outputs*, sendo estes determinados apenas pelo estado em que o sistema se encontre. Insinuando, por isso, que para ocorrer uma mudança no valor do *output* é necessário a ocorrência de uma mudança do estado atual do sistema.

### 2.1.1.2 Máquinas de *Mealy*

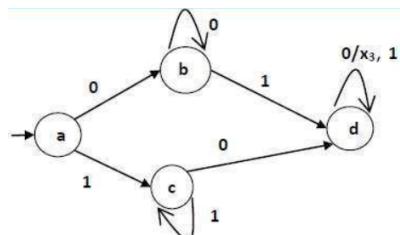


Figura 2.2: Modelação de uma máquina de *Mealy* [6]

Nomeada em homenagem ao seu inventor *George H. Mealy*, nas máquinas de *Mealy* as transições entre estados e o próprio estado ativo são os responsáveis por produzir os *outputs*. Esta característica, resulta, geralmente, na redução do número de estados da máquina, pois grande parte da lógica pode ser introduzida nas transições. Para além disso, estas máquinas apresentam tempos de reação aos *inputs* superiores às máquinas de *Moore*.

### 2.1.2 Modelação tabelar

Também é possível modelar uma máquina de estados utilizando uma tabela que contenha os estados para os quais o sistema pode transitar, consoante o estado atual do sistema e os seus *inputs*.

A aplicação desenvolvida utiliza esta abordagem, permitindo associar à tabela o estado atual do sistema, o evento que dipara a transição, o próximo estado, bem como três tipos de ações associadas ao estado e uma associada à transição. Para além disso, a *UI* desenvolvida permite associar outros tipos de características à máquina de estados, tais como o conceito de ortogonalidade, alteração de valores dos pinos do microprocessador/ microcontrolador ou a execução de ações hierárquicas, características essas que serão abordadas no capítulo 3.

### 2.1.3 Modelação hierárquica

Embora as máquinas de *Moore* e *Mealy* permitam a resolução de problemas de pequena complexidade, o facto deste formalismo envolver repetições, já que múltiplos eventos são lidados de forma semelhante em estados diferentes, faz com que o número de estados presente numa máquina aumente exponencialmente com o aumento da complexidade do sistema que se pretende controlar. Tal acontece porque os formalismos convencionais não têm mecanismos que permitam generalizar comportamento comum, levando à repetição de ações e transições em diferentes estados.

#### 2.1.3.1 Harel statecharts

Surgiu, por isso, nos anos 80, o conceito de *statechart* ou *hierarchical state machine*, nos quais, os conceitos de hierarquia, concorrência e comunicação vieram trazer mais profundidade a este tipo de formalismo, mantendo simultaneamente os diagramas bem estruturados e fazendo com que diagramas de pequena dimensão consigam modelar comportamentos complexos. [7]

“De forma a ser útil, uma máquina de estados deve ser modular, hierárquica e bem estruturada.”

—David Harel

Os conceitos de regiões ortogonais, estados compostos capazes de serem executados paralelamente, eventos que permitem descrever comportamentos complexos, utilização de condições para a ocorrência de transições, lógica temporal, transições entre nível, bem como ações à entrada e à saída de estados foram alguns dos conceitos que permitiram a aplicação das máquinas de estados a problemas reais, de elevada complexidade.

### 2.1.3.2 UML statecharts

Hoje em dia, a topologia utilizada para modelar uma máquina de estados passa pela utilização do conceito de *UML statecharts*, baseados na notação dos *Harel statecharts*, mas estendendo a sua notação introduzindo princípios orientados a objetos, o que permitiu remover as limitações das máquinas de estados finitas tradicionais, mas retendo os seus principais benefícios, suportando ações que dependem do estado atual do sistema e do evento que ativa a transição, características das máquinas de *Mealy*, mas também suportando ações que ocorrem à entrada ou à saída de um estado, associadas apenas ao estado e não à transição, características das máquinas de *Moore*.

A modelação em *UML* preserva a representação tradicional dos diagramas de estado onde "nós" representam estados e conetores entre estes representam transições entre estados.

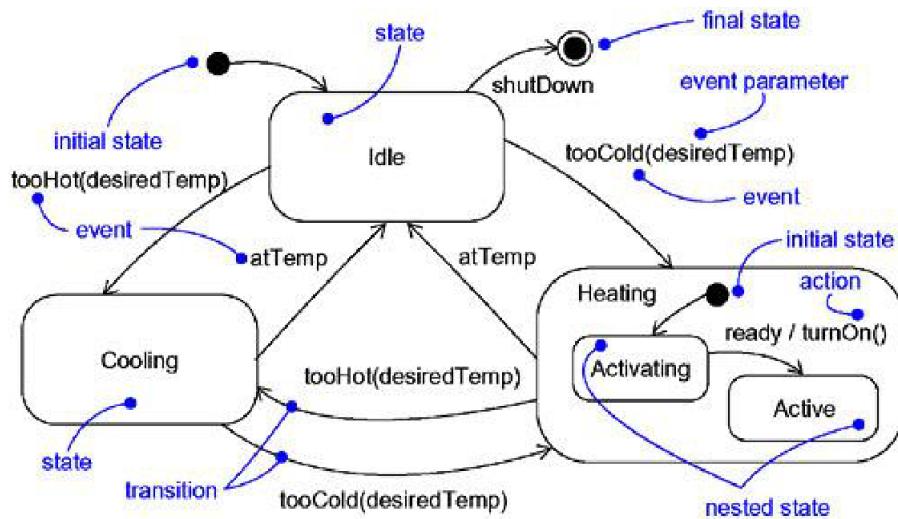


Figura 2.3: Representação gráfica de estados, transições, eventos e ações em *UML* [8]

A indicação do estado inicial é um requisito obrigatório, sendo representada através de uma transição que parte de um círculo preenchido que não pode conter eventos a si associados, mas pode conter ações.

Um estado é constituído pela *string* que corresponde ao seu nome e por uma série de funcionalidades que podem assumir a forma de:

- "*Entry*" ou "*exit actions*" descritas posteriormente às instruções "*entry /*" ou "*exit /*", respetivamente. Este tipo de ações devem ser utilizadas quando se pretenda executar uma ação sempre que ocorra uma entrada ou saída do estado, independentemente de qual a transição que levou à entrada ou saída deste;
- "*Internal transitions*" , a sua representação consiste na indicação do evento que provoca a transição, seguido de um caractere "/" e da ação associada a essa mesma transição. São utilizadas em situações em que se pretenda responder a um evento executando uma ação, mas sem que ocorra a saída do estado atual. Este tipo de ação não poderia ser executada utilizando ações associadas a *self-transitions*, pois nesses casos a transição provoca a saída

do estado, o que leva também à execução da *exit action*, seguida da ação da *self-transition*, e por fim da *entry action* do estado;

- "Activity actions", representadas posteriormente à instrução "*do /*". São utilizadas quando se pretende a execução de uma ação continuamente, durante o tempo em que o estado se encontre ativo;
- "Deferred events" a sua representação consiste na indicação do evento a ser diferido, seguido da instrução "*/ defer*". São utilizados sempre que se pretenda reconhecer a ocorrência de um evento posteriormente a este ter acontecido, ou seja, um estado que contenha este tipo de evento ignora a sua ocorrência. No entanto, essa informação é gravada e após a transição para um estado que contenha esse evento, mas não estando este diferido, a transição que corresponda a esse evento é disparada mesmo este não tenha ocorrido nesse instante.

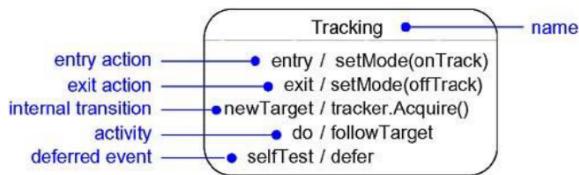


Figura 2.4: Representação gráfica de um estado e suas funcionalidades em UML [8]

As transições são representadas com setas e legendadas com base nos eventos que as ativam, podendo, opcionalmente, conter ações que são executadas na sua ocorrência. Estas podem ter a si associadas *guard transitions* que são expressões booleanas contidas em parêntesis retos colocadas adiante da indicação do evento que dispara a transição. Uma vez ocorrido o evento, o valor dessa expressão é avaliado, disparando a transição no caso da condição se verificar.

Uma *triggerless transition*, também chamada de *completion transition* é representada sem qualquer legenda, e é disparada quando a atividade do seu estado de origem é completada.

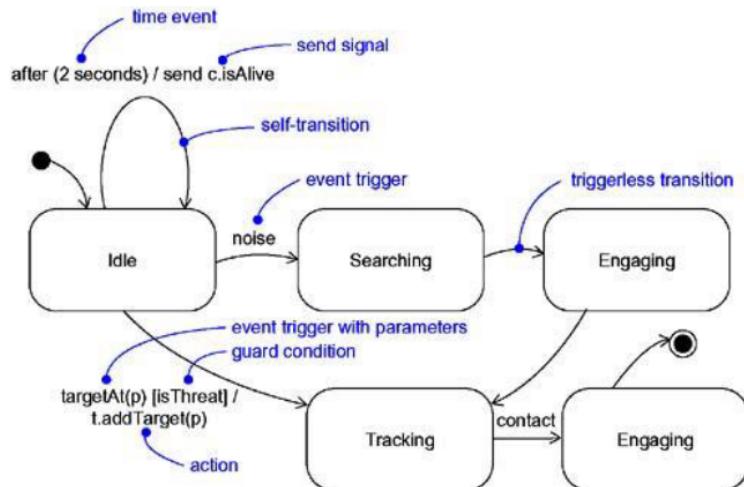


Figura 2.5: Representação gráfica de uma transição e suas funcionalidades em UML [8]

A utilização destas funcionalidades, aliadas ao conceito de *hierarchically nested states* tornaram as máquinas de estado modeladas em *UML* mais poderosas, onde um estado que esteja contido dentro de outro é denominado de *substate* e o estado que o contém de *composite state*. Quando o sistema se encontra num *substate*, encontra-se simultaneamente no estado que o contém, este reagirá aos eventos sob a perspectiva do *substate* e caso não tenha a si associada forma de reagir a um determinado evento, este é lido pelo seu nível superior. Sendo assim, os *substates* precisam apenas de ter definidas as diferenças em relação aos *superstates*, pois herdam o comportamento do seu nível superior.

O conceito de ortogonalidade permite que um estado contenha duas ou mais regiões ortogonais, e a presença desse estado implica também estar presente, simultaneamente, em todas as suas regiões ortogonais independentes e concorrentemente ativas. Estas regiões podem comunicar, sincronizar os seus comportamentos e enviar eventos entre si.

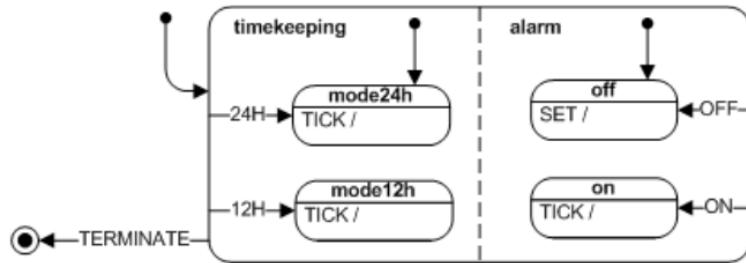


Figura 2.6: Representação gráfica de regiões ortogonais em *UML* [9]

## 2.2 Sistemas Embebidos

Um sistema embebido consiste numa combinação de *hardware* e *software* que foi desenhado para efetuar tarefas específicas [1]. Uma parte essencial da sua constituição é a fonte de alimentação que geralmente é de 5V, mas também pode admitir outros valores inferiores como 3.3 ou 1.8V.

O processador do sistema pode assumir a forma de microprocessador ou microcontrolador e ter diversas arquiteturas como 8-bit, 16-bit e 32-bit. A escolha do processador adequado depende de vários critérios como velocidade, preço por unidade, número de pinos *I/O* e ferramentas de *software* disponíveis. Um microcontrolador contém o elemento de processamento, mas também outras funcionalidades como as unidades de *RAM*, *ROM*, *timer-counters* ou portas de comunicação, todas elas incorporadas num único *chip*. Por outro lado, um microprocessador contém apenas a unidade de processamento, o que significa que para completar o sistema é necessária a adição de elementos básicos como a *RAM* e a *ROM* de forma externa ao processador.

Os tipos de memória podem assumir a forma de *Random Access Memory(RAM)*, *Read-Only Memory(ROM)* ou *Electrically Erasable Read-Only Memory (EEPROM)*.

Existem vários tipos de portas de comunicação como a *UART*, *CAN*, *SPI*, *I2C*, *USB*, *Ethernet*, *RS-232*, *RS-432* ou a *RS-485*. [10]

Podem assumir a forma de *ready-made boards* que vão desde o nível simples até ao avançado, de acordo com as necessidades da aplicação e permitem reduzir o tempo de implementação necessário. Alguns exemplos passam pela: *Arduino Board*, *Raspberry Pie*, *Banana Pie*, *Intel Galileo* ou *Beaglebone*. Em alternativa podem ser utilizados *Application Specific Integrated Circuits (ASICs)* ou *Field Programmable Gate Arrays (FPGAs)* que apresentam características de redução de custo e tamanho pequeno. [11]

Com recurso a ferramentas como *assemblers* ou compiladores é possível efetuar o desenvolvimento e implementação de código no *software* de um sistema embebido.

## 2.3 Linguagens utilizadas para programação embebida

Nem todas as linguagens são adequadas para este tipo de sistemas, pois a sua programação requer acesso ao baixo nível do sistema que se pretende controlar, bem como a utilização do mínimo de recursos possível.

A linguagem mais popular para programação embebida é a linguagem *C*, já que permite o acesso aos componentes do nível mais baixo do sistema, aliada à baixa utilização de memória que esta requer e à sua velocidade de execução.

No entanto, programadores mais experientes, tendencialmente, optam pela utilização do *C++*, pois tal como o *C*, é uma linguagem de execução rápida que, para além de oferecer acesso aos recursos de baixo nível do sistema que controla, permite também uma abordagem de programação orientada a objetos.

A estabilidade e o grande número de ferramentas e bibliotecas de desenvolvimento que a linguagem *Java* oferece, faz com que esta também seja bastante popular. Os programas escritos em *Java* são independentes da máquina onde são implementados, pois são executados dentro da *JVM (Java Virtual Machine)*, o que faz com que um programa escrito em *Java* seja facilmente extensível e portável.

Mais recentemente, após se ter estabelecido como uma das linguagens de eleição na área do desenvolvimento *Web*, a linguagem *Python* tem vindo a ganhar terreno no mundo da programação embebida. O seu estilo de codificação concisa e de fácil interpretação permite a escrita rápida de programas e torna-a uma das linguagens mais populares para novos programadores.

Linguagens como o *Rust*, *Ada*, *JavaScript*, *Go*, *Lua*, *Assembly*, *Verilog* ou *VHDL* também são linguagens bastante utilizadas para programação de *software* embebido. [12]

## 2.4 Métodos de estruturação de uma máquina de estados em C

A utilização de máquinas de estados é um dos padrões mais populares para desenho de aplicações em sistemas embebidos. Tal advém do facto deste ser um dos métodos mais eficazes para desenvolvimento de código orientado a eventos, isto é, código a ser implementado em sistemas que esperam continuamente pela ocorrência de um evento externo que altera o comportamento do sistema.

Os sistemas embebidos são candidatos perfeitos para a implementação deste tipo de técnicas, pois são destinados a realizar tarefas predefinidas, com programas que devem sequenciar uma série de ações ou lidar com *inputs* de forma diferente consoante a fase de funcionamento em que se encontram.

Na perspetiva da programação embebida, a codificação do comportamento de um sistema de eventos discretos consiste na criação de código que execute respostas a determinados eventos, eventos esses que poderão ter origem em interrupções do sistema, na indicação do expirar de um *timer* ou no valor de um *input* do sistema. Essa resposta não depende apenas da natureza do evento, mas também do histórico de eventos passados em que o sistema esteve envolvido.

Nesse contexto, a tentativa de implementação deste tipo de código sem a utilização de nenhuma regra de estruturação leva, geralmente, ao aparecimento de grandes ramificações, densas e complexas de código, com um grande volume de instruções condicionais como o "*if-else*" ou "*switch-case*" acompanhadas de um número elevado de *flags*, que tem como consequência a elaboração de código confuso, pouco estruturado e de difícil interpretação, gerando aquilo que é vulgarmente denominado por "*spaghetti code*". [13]

As máquinas de estados constituem uma parte muito importante do mundo da programação embebida, já que, quando utilizadas corretamente, seguindo um conjunto de técnicas, permitem reduzir de forma significativa a acumulação de expressões condicionais, diminuir a probabilidade da ocorrência de erros, tornando o código mais perceptível e intuitivo ao olho humano.

Um fator comum a todos os diferentes métodos de codificação de uma máquina de estados, em C, é a utilização de uma variável denominada por "*estado*", cujo valor define a forma da máquina reagir a eventos. Esta variável assume, geralmente, valores predefinidos fixados por uma enumeração e a sua existência permite reduzir drasticamente o número de ramificações necessárias, pois torna desnecessário a verificação do histórico de eventos ocorridos no passado, eliminando a necessidade de identificação do contexto de execução no código, passando apenas a ser necessário efetuar o teste de uma única variável e não de muitas, suprimindo a necessidade de recorrer a uma lógica condicional excessiva, visto que, a ocorrência de um evento passa a estar associada ao "*estado*" atual do sistema.

As abordagens mais comuns para implementação de uma máquina de estados orientada a eventos, passam pela utilização de expressões condicionais como o "*switch-case*" ou o "*if-else*", ou pela utilização de *arrays* de apontadores para funções que executam o comportamento das chamadas *jump tables*.

De seguida serão apresentadas análises ao desempenho e vantagens de cada uma destas soluções, bem como exemplos da sua aplicação, em C, em programação embebida.

É importante referir que neste capítulo da dissertação, as abordagens de codificação mencionadas conterão apenas as instruções de código que explicitam a estruturação da máquina de estados, isto é, os estados que a definem e os eventos que provocam uma transição de estados visto que estes comportamentos são comuns a todos os formalismos de modelação vistos na secção 2.1.

Contudo, se uma máquina de estados só realizasse a função de estruturação dos estados do sistema, esta não traria utilidade na prespetiva de um programador, já que a existência de ações é que traz esse fator.

Existem vários tipos de ações que variam consoante o formalismo de modelação de uma máquina de estados que está a ser utilizado, podendo estas ocorrer, por exemplo, à entrada e à saída de um estado, ou mesmo associadas à ocorrência de eventos que provoquem a sua transição. Sendo assim, neste capítulo, não será abordada a forma de codificação dos vários tipos de ações, nem quais os formalismos de modelação que a aplicação desenvolvida permite associar à máquina de estados, encontrando-se, no entanto, tal explicitado no capítulo 3.

#### 2.4.1 Expressões condicionais

Uma abordagem utilizando expressões condicionais é uma solução simples e intuitiva, na qual são utilizadas instruções "if-else" ou "switch-case" que executam uma verificação estado a estado, sendo introduzidas dentro de cada uma dessas expressões outras expressões condicionais que efetuam a verificação da execução de um evento, ocorrendo uma transição de estados caso tal condição se verifique. Tomando como exemplo a seguinte máquina de estados:

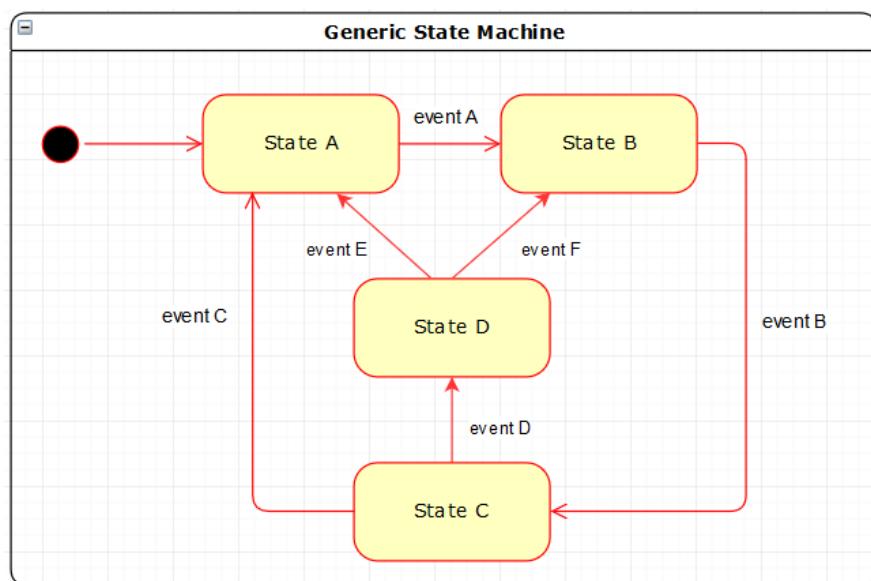


Figura 2.7: Modelação de uma máquina de estados genérica em UML

Uma possível implementação, em C, utilizando uma abordagem *switch-case* seria:

```
//Declare the possible states
typedef enum
{
    stateA,
    stateB,
    stateC,
    stateD,
} stateNames;

//Declare the initial state of the state machine
stateNames currentState = stateA;

void runSM()
{
    switch(currentState)
    {
        case stateA:
            if(eventA())
                currentState = stateB;
            break;

        case stateB:
            if(eventB())
                currentState = stateC;
            break;

        case stateC:
            if(eventC())
                currentState = stateA;
            if(eventD())
                currentState = stateD;
            break;

        case stateD:
            if(eventE())
                currentState = stateA;
            if(eventF())
                currentState = stateB;
            break;

        default: break;
    }
}
```

Figura 2.8: Implementação de uma máquina de estados, em C, utilizando *switch-case*

A sua implementação, utilizando uma abordagem *if-else* não seria muito diferente:

```
//Declare the possible states
typedef enum
{
    stateA,
    stateB,
    stateC,
    stateD,
} stateNames;

//Declare the initial state of the state machine
stateNames currentState = stateA;

void runSM()
{
    if(currentState == stateA)
    {
        if(eventA())
            currentState = stateB;
    }
    else if(currentState == stateB)
    {
        if(eventB())
            currentState = stateC;
    }
    else if(currentState == stateC)
    {
        if(eventC())
            currentState = stateA;
        if(eventD())
            currentState = stateD;
    }
    else if(currentState == stateD)
    {
        if(eventE())
            currentState = stateA;
        if(eventF())
            currentState = stateB;
    }
    else{}
}
```

Figura 2.9: Implementação de uma máquina de estados, em C, utilizando *if-else*

Quando o número de estados e de eventos do sistema é pequeno, a utilização de instruções condicionais permite, a um utilizador que analise o código, uma percepção rápida do comportamento do sistema. No entanto, quando o número de estados e de eventos aumenta, o código que os descreve fica naturalmente confuso, já que, tomando como exemplo uma abordagem utilizando *switch-cases* numa máquina que contenha trinta estados, contendo cada um deles instruções de vinte linhas, é facilmente perceptível que o seu código será extenso e repetitivo, tornando a sua legibilidade difícil, bem como a sua manutenção. Para além disso, o esquecimento de uma simples instrução *break* comprometeria o funcionamento de todo o sistema.

À parte dos problemas mencionados, a sobre-utilização de expressões condicionais resulta no aumento do tempo de execução do programa, o que resulta numa desvantagem significativa.

Um compilador convencional tem, tipicamente, três formas de implementar uma instrução *switch*: pode interpretá-lo como uma sequência de *if-elses*, pode utilizar as chamadas *jump tables*, ou uma abordagem híbrida de ambas. O compilador decide qual a abordagem a utilizar através dos valores atribuidos aos *cases*, caso estes sejam contíguos, são utilizadas *jump tables*, caso estes sejam dispersos então a abordagem *if-else* é mais provável de ser utilizada. Embora alguns compiladores permitam a escolha do método de implementação, em grande parte dos casos o utilizador encontra-se dependente do algoritmo heurístico de escolha do compilador. Sendo assim, a abordagem via *jump-tables* não é sempre executada.

#### 2.4.2 Apontadores para funções

A utilização de apontadores para funções permite assegurar que o compilador faz o que o utilizador pretende, já que uma abordagem orientada a apontadores para funções baseia-se na implementação de *jump tables*, que são métodos eficientes de lidar com eventos em software, a sua utilização permite aumentar a estabilidade e desempenho do programa, obtendo tempos de execução consistentes e rápidos. [14]. É recomendável a sua utilização quando a máquina de estados apresenta muitos estados ou o código associado a cada estado seja elevado. Utilizando a máquina de estados da figura 2.7 como modelo a implementar, uma possível codificação, em C, utilizando uma abordagem utilizando apontadores para funções seria:

```

//Declare possible states
typedef enum
{
    stateA,
    stateB,
    stateC,
    stateD,
    NUM_STATES
} stateNames;

//Declare initial state
stateNames currentState = stateA1

void runstateA();
void runstateB();
void runstateC();
void runstateD();

struct stateStruct
{
    stateNames name;
    void(*funcPoint)();
};

stateStruct array[]=
{
    {stateA, runstateA},
    {stateB, runstateB},
    {stateC, runstateC},
    {stateD, runstateD},
};

void runstateA()
{
    if(eventA())
        currentState = stateB;
}

void runstateB()
{
    if(eventB())
        currentState = stateC;
}

void runstateC()
{
    if(eventC())
        currentState = stateA;

    if(eventD())
        currentState = stateD;
}

void runstateD()
{
    if(eventE())
        currentState = stateA;

    if(eventF())
        currentState = stateB;
}

void runSM()
{
    (*array[currentState].funcPoint)();
}

```

Figura 2.10: Implementação, em C, de uma abordagem utilizando apontadores para funções

Tal como nas implementações anteriores, inicialmente é realizada uma enumeração dos estados possíveis da máquina de estados assumir. No entanto, neste caso são utilizados *arrays* que contêm apontadores para funções, onde o seu índice assume o valor da variável "estado"enumerada e cada uma das funções corresponde ao comportamento de um estado da máquina.

## 2.5 Softwares semelhantes

De forma a desenvolver uma aplicação competitiva com o que já existe no mercado, foi realizada uma pesquisa sobre as principais aplicações existentes, seguida de uma análise das suas principais funcionalidades.

Algumas delas permitem modelar uma máquina de estados de forma tabelar, enquanto outras permitem-no fazer graficamente. No entanto, nem todas as aplicações de modelação incluem a funcionalidade de interpretação e geração do código equivalente ao modelo especificado e, dentro das que o permitem, nem todas se encontram orientadas para a geração de código para *software* embebido. Sendo esse o principal foco da aplicação a desenvolver nesta dissertação, serão apenas mencionadas as principais aplicações que contêm essa funcionalidade:

### 2.5.1 *QM<sup>TM</sup> modelling tool*

Esta ferramenta, desenvolvida pela *Quantum Leaps*, consiste num *software* gratuito que permite modelar uma máquina de estados graficamente, segundo a norma *UML* para *statecharts*, bem como a obtenção de forma automática do código, em *C* ou *C++*, para ser implementado em *software* embebido.

O código gerado é orientado a objetos, o que significa que é fundamentalmente composto por classes utilizando os conceitos de herança e polimorfismo. [15] A ferramenta foi desenvolvida com uma *code-centric approach*, dando ao utilizador controlo total sobre a estrutura do código gerado, nome dos diretórios e elementos que entram em todos os ficheiros gerados, bem como a possibilidade de inclusão do próprio código no código sintetizado. [16] Este *software* foi escolhido por centenas de empresas para modelação de produtos electrónicos aplicados a uma grande variedade de mercados, nas quais estão incluídas empresas como a *Bosch*, *Hitachi*, *NASA* ou a *Toshiba*. [17]

Encontra-se disponível para utilizadores *Windows*, *Linux* e *MacOS*.

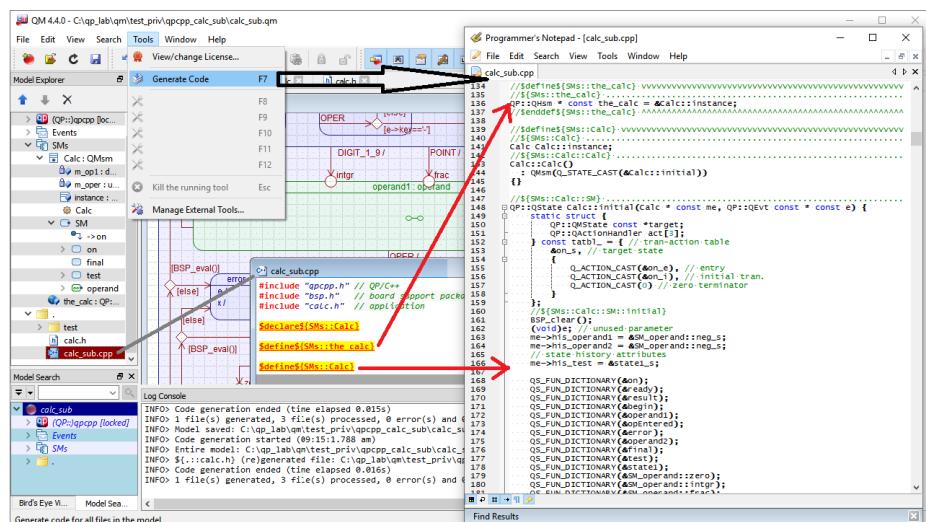


Figura 2.11: Interface do software *QM<sup>TM</sup>* [18]

### 2.5.2 Simulink

Através da utilização da biblioteca "stateflow" no *Simulink*, é possível modelar *state transition diagrams*, *flow charts*, *state transition tables* e *truth tables*.

Permite gerar código em *C* ou *C++* usando o *Simulink Coder*<sup>TM</sup>, *VHDL* e *Verilog* para ser implementado em *FPGAs* ou *ASICs* usando o *HDL Coder*<sup>TM</sup> ou *ST* da norma *IEC 61131-3* para ser implementado em *PLCs* ou *PACs* usando o *Simulink PLC Coder*<sup>TM</sup>. [19]

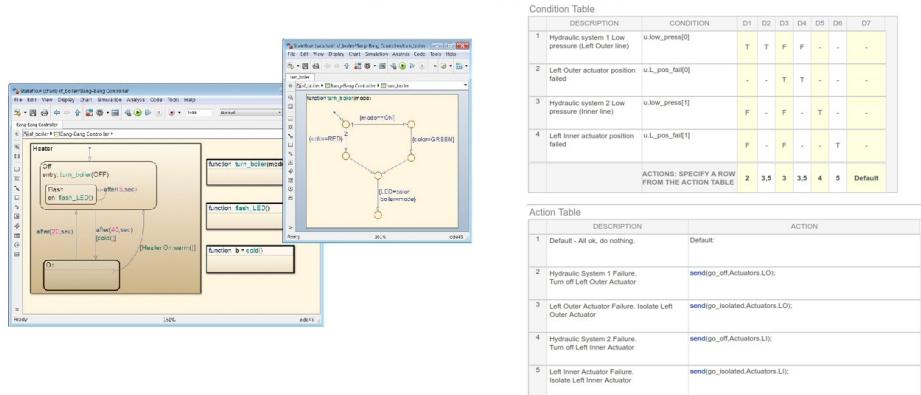


Figura 2.12: Interface da biblioteca stateflow em Simulink [19]

### 2.5.3 IAR Visual State

Esta ferramenta, desenvolvida pela *IAR Systems*, permite o desenho e geração, em *Windows*, de código de máquinas de estados modeladas segundo a norma *UML*. O código é gerado de forma bastante compacta nas linguagens C, C++, C# ou Java. Disponibiliza um suporte de *debug* do hardware, com *feedback* direto da máquina de estados desenhada, podendo ser vistos instantaneamente quais os estados ativos, bem como quais as transições que provocaram a sua transição. Inclui a funcionalidade de validação e verificação dos modelos, bem como a geração automática de documentação. [20]

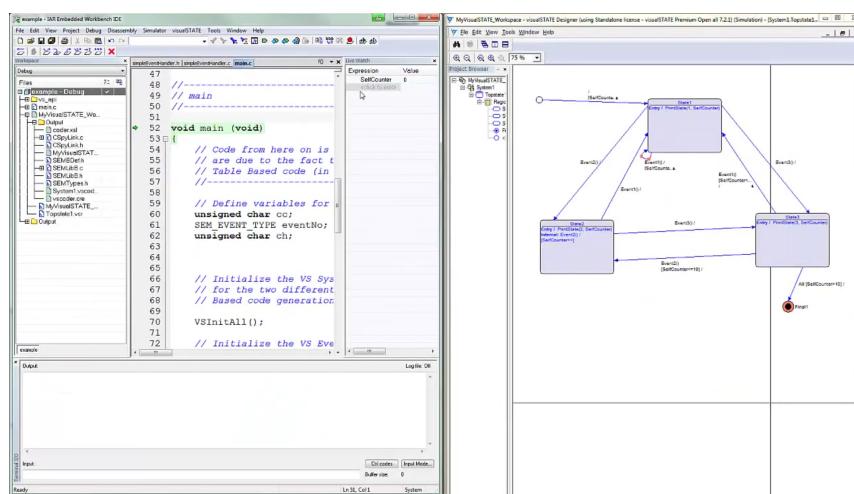


Figura 2.13: Interface do software IAR Visual Studio [21]

### 2.5.4 YAKINDU Statechart Tools

Esta ferramenta pode ser utilizada na plataforma *Eclipse* e permite a modelação, validação, edição e simulação de *statecharts* [22], contendo também a funcionalidade de conversão do modelo para um ficheiro *SCXML*. Permite gerar código para *C*, *C++*, *Java*. A possibilidade de gerar código para *Python*, *Swift* e *TypeScript* também é possível, notando que estas funcionalidades se encontram numa fase *beta*.

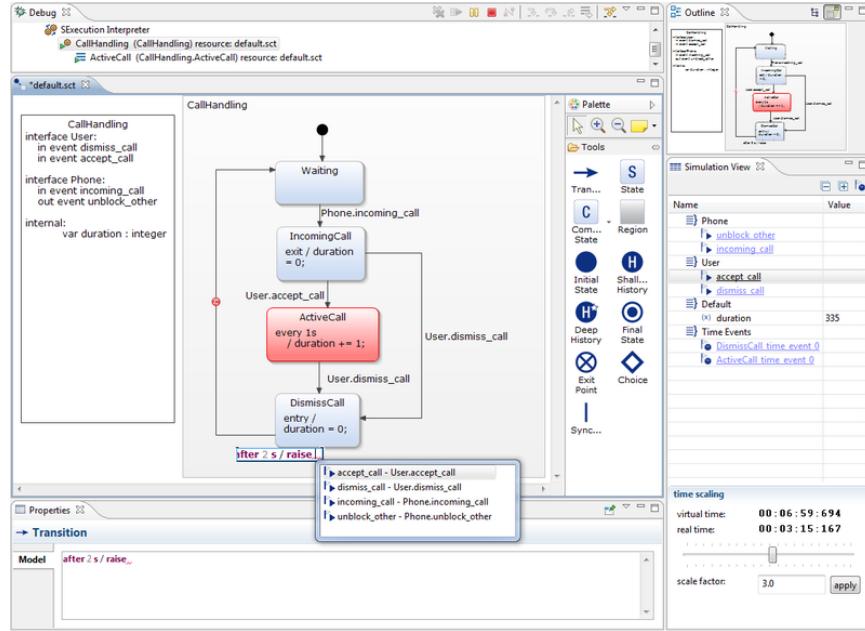


Figura 2.14: Interface da ferramenta *YAKINDU Statechart Tools* no *IDE Eclipse* [22]

### 2.5.5 Outros

Softwares como o *StarUML* [23] da *MKLabs*, o *Visual Paradigm* [24] da *Visual Paradigm International Ltd.* e o *Lab View* [25] da *National Instruments* também permitem a modelação de forma gráfica de uma máquina de estados e a sua conversão para diferentes linguagens de código. No entanto, não são tão orientados para *software* embebido como os softwares mencionados anteriormente.

Aplicações mais simples como o *SM Cube* [26], *SMC* [27], *Ragel* [28] ou o *StateForge* [29] também permitem executar essa funcionalidade.



# Capítulo 3

## Desenvolvimento

O objetivo da presente dissertação consistiu no desenvolvimento de uma aplicação para *Windows* que permitisse ao utilizador obter de forma simples e automática o código para sistemas embebidos correspondente a um modelo de máquinas de estados. Para tal foram utilizadas diversas tecnologias, podendo ser visto na figura 3.1 as diferentes fases do desenvolvimento do projeto, bem como as linguagens que foram utilizadas para a sua elaboração.

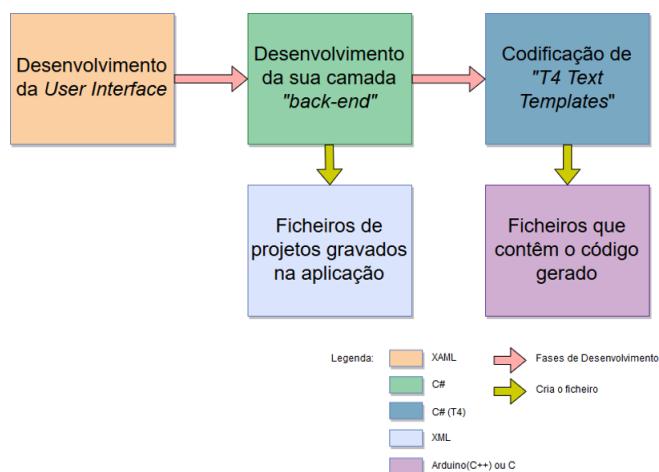


Figura 3.1: Diagrama das diferentes fases de desenvolvimento do projeto

Numa fase inicial do desenvolvimento do projeto foram estudadas quais as ferramentas a utilizar que permitissem desenvolver estas funcionalidades.

### 3.1 Tecnologias

Os dois grandes componentes que constituem a solução concebida passam pelo desenvolvimento de uma *UI* que permite modelar máquinas de estado e que assume a forma de *desktop application* para *PCs Windows*, e pela geração dos ficheiros de texto que contêm o código relativo a essas máquinas. Deste modo, a escolha do ambiente de desenvolvimento e das suas plataformas deveria ir de encontro à potencialização da eficiência na criação destas funcionalidades.

### 3.1.1 Plataformas de desenvolvimento de uma *UI* para *Windows*

Com o intuito de criar uma aplicação para *Windows*, foi necessário escolher a plataforma para desenvolvimento dessa mesma aplicação. A *Microsoft* fornece quatro plataformas que permitem a criação de aplicações para ser executadas em *Windows*.

O *UWP*, *WPF* e *Windows Forms* disponibilizam um *managed runtime environment*. O *Windows Runtime*, no caso do *UWP*, o e o *.NET* para o *Windows Forms* e *WPF*. Estas *frameworks* aliam ferramentas de *design* visual com linguagens de *markup* que permitem criar de forma rápida e estruturada uma *UI*.

O *UWP*, lançado no mercado em 2015, é a plataforma que lidera a criação de aplicações para *Windows 10*, é altamente customizável e utiliza a linguagem de marcação *XAML* para separar a camada de apresentação da camada lógica. Esta *framework* é utilizada em aplicações que necessitem de *UIs* sofisticadas com funcionalidades gráficas poderosas e com grande nível de customização.

O *WPF*, tal como a plataforma anterior, utiliza o *XAML* para separar a camadas de apresentação da camada lógica, tendo acesso total à *framework .NET*. As técnicas de desenvolvimento são bastante semelhantes às utilizadas na *UWP*, portanto a curva de aprendizagem entre estas é pequena.

O *Windows Forms* é a plataforma original para desenvolvimento de aplicações para *Windows*, permite a criação de uma *UI* leve e tem acesso total à *framework .NET* não utilizando, no entanto, *XAML*.

A *Win32 API*, também chamada de *Windows API* é utilizada para aplicações que necessitam de acesso direto ao *Windows* e *hardware*. Ao contrário das três frameworks anteriores, não necessita de *managed runtime environments* como o *.NET* ou o *WinRT*, o que a torna a plataforma de eleição para aplicações que necessitam alto nível de performance ou acesso direto ao *hardware* do sistema. [34]

### 3.1.2 Escolha da plataforma e do *IDE*

Tendo em conta as características anteriores, a escolha da plataforma a utilizar foi feita entre o *UWP* e o *WPF*. O facto das aplicações geradas em *UWP* só poderem ser executadas em dispositivos cujo sistema operativo seja o *Windows 10*, contrariamente às geradas em *WPF* que podem ser executadas em *Windows 7* ou superior, bem como o facto das funcionalidades extra que a sua utilização traz, em detrimento da utilização do *WPF* não serem críticas para o desenvolvimento do projeto em questão, aliadas à existência de um maior número de documentação de apoio à utilização do *WPF*, já que o *UWP* é uma plataforma mais recente, levaram à escolha do *WPF* como plataforma de desenvolvimento da *UI* do projeto.

O *WPF*, *Windows Presentation Foundation*, inclui elementos como a *Extensible Application Markup Language (XAML)* que permitem efetuar o design da *UI* de forma estruturada, separada do código *back-end* que define o comportamento da aplicação e que interage com a linguagem *markup* através de definições de classes parciais.

O *XAML* representa a instanciação de objetos de certos tipos definidos em *assemblies*, distingue-se da maioria de outras linguagens *markup* pelo facto destas serem, tipicamente, linguagens interpretativas, sem uma ligação tão direta à camada *back-end*. Tomando como exemplo o botão que permite gerar o código na aplicação desenvolvida:



Figura 3.2: Design do botão "Generate Code"

Este é criado em *XAML* utilizando a seguinte instrução:

```
<Button x:Name="generateCode" Content="Generate Code" Click="GenerateCode_Click"
|   Width="125" Height="50" FontSize="18" Foreground="White" Background="#21ba45"/>
```

Figura 3.3: Instruções em "XAML" para gerar o botão

Uma vez efetuado o "click" nesse mesmo botão, o método "*GenerateCode Click*" que se encontra declarado na camada *back-end* da aplicação, em *C#*, será executado:

```
//GENERATE CODE EVENT for selected SM, sends all the info to the TextTemplatePreProcessor File
public void GenerateCode_Click(object sender, RoutedEventArgs e){...}
```

Figura 3.4: Método "GenerateCode Click"

Que, por sua vez, irá verificar se todos os parâmetros necessários à geração se encontram especificados e irá gerar o código, apresentando uma indicação de tal:

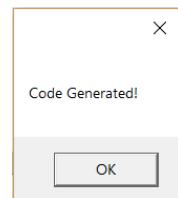


Figura 3.5: MessageBox que surge após click do botão

O *Microsoft Visual Studio* é um *Integrated Development Environment (IDE)* desenvolvido pela *Microsoft*, que inclui plataformas de desenvolvimento de *software* como a *Windows API*, o *Windows Forms*, o *WPF* e o *Microsoft SilverLight*. Este *IDE* suporta 36 linguagens, nas quais estão incluídas o *C*, *C++*, *C#*, *Visual Basic .NET*, *JavaScript*, *XML*, *HTML* e *CSS*.

Tal como foi justificado anteriormente, o *WPF* foi escolhido como plataforma de desenvolvimento da *UI*. Sendo assim, a escolha mais adequada de *IDE* para desenvolvimento do projeto passou pela utilização da versão gratuita e mais recente do *Microsoft Visual Studio Community 2017*.

Numa fase avançada do desenvolvimento do projeto, em Abril de 2019, a *Microsoft* lançou a versão 2019 do *Visual Studio*, não tendo, no entanto, sido utilizada essa versão do *IDE*.

### 3.1.3 Geração de código utilizando *T4 Text Templates*

Uma vez definido o ambiente de desenvolvimento e a plataforma de criação da *UI*, a próxima decisão a ser tomada passava pela forma de geração do ficheiro de texto que contém o código a ser implementado no sistema embebido e que cujo conteúdo deveria ser variável consoante as especificações do utilizador na *UI*.

Um *T4 Text Template* utiliza uma mistura de blocos de texto e blocos de controlo lógico para gerar um ficheiro de texto. Os blocos de controlo lógico, utilizam fragmentos de código em *Visual C#* ou *Visual Basic* e possibilitam a repetição de secções de texto, através da utilização expressões condicionais ou de instruções como o "for" e o "while".

Tomando como exemplo um dos ficheiros T4 elaborados, uma secção do seu código, em *C#*, consiste na geração das instruções de enumeração dos estados que constituem as máquinas de estados modeladas. A seguinte instrução num *T4 Text Template* geraria essa mesma enumeração:

```
<#@ template debug="true" hostspecific="false" language="C#" #>
<#@ output extension=".c" #>

//Declare possible states
typedef enum
{
    stateA,
    stateB,
    stateC,
    NUM_STATES
} stateNames;
```

Figura 3.6: *T4 Text Template* sem blocos lógicos

Contudo, a geração de texto sem a utilização de blocos lógicos não apresenta uma mais valia, pois o objetivo pretendido passa pela geração de um ficheiro que seja variável e dinâmico consoante as especificações de um utilizador. Assim sendo, a instrução seguinte permite gerar o conteúdo equivalente à instrução anterior, fazendo-o, no entanto, de forma dinâmica:

```
<#@ template debug="true" hostspecific="false" language="C#" #>
<#@ output extension=".c" #>
<#@ parameter type="System.Collections.Generic.List`1[String]" name="allStatesList" #>

//Declare possible states
typedef enum
{
    @<#foreach(string item in allStatesList){ #>
        <#=item#>,
    @<# } #>
    NUM_STATES
} stateNames;
```

Figura 3.7: *T4 Text Template* com blocos lógicos

No exemplo da Figura 3.7, é possível encontrar os três tipos de blocos que foram utilizados nos *T4 Text Templates*.

Os elementos diretivos encontram-se entre as instruções `<#@` e `#>` e permitem controlar a forma como o *template* é processado.

Os blocos de texto não contêm nenhuma instrução associada e o seu conteúdo é diretamente copiado para o ficheiro de texto.

Os blocos de controlo que podem ser do tipo *standart* ou *expression control blocks*, sendo do primeiro tipo quando se encontram entre `<#` e `#>`, consistindo em secções de código utilizadas para gerar parte do ficheiro de texto, passando a ser do segundo tipo caso se encontrem entre `<#=` e `#>`, tendo como função obter o valor da expressão em si contida, convertê-la numa *string* e inseri-la no ficheiro de texto a ser gerado. [35]

Utilizando este tipo de instruções, aliando blocos de texto e blocos lógicos codificados em *C#*, foi possível programar uma lógica de geração ficheiros de texto que contenham o código correspondente à modelação efetuada na *UI*.

Estes *templates* foram codificados de modo a interpretarem de forma correta as informações provenientes da *UI*, que guarda os dados introduzidos pelo utilizador em listas e envia-as para os *templates* quando o método "*GenerateCode Click*" é executado, através das seguintes instruções de código, em *C#* que se encontram presentes na camada lógica da aplicação:

```
// Get the current directory.
string path = Directory.GetCurrentDirectory() + "\\generatedCode";
if (!Directory.Exists(path)) // if it doesn't exist, create
    Directory.CreateDirectory(path);

//Generate the main file
Template2 t = new Template2();
t.Session = new Microsoft.VisualStudio.TextTemplating.TextTemplatingSession();
t.Session["stateMachineList"] = stateMachineList;
t.Session["allOutsList"] = allOutsList;
t.Session["allOutsTypeList"] = allOutsTypeList;
t.Session["allTransList"] = allTransList;
t.Session["allTransTypeList"] = allTransTypeList;

t.Session["typeOfTransformation"] = comboBoxTypeTransformation.Text;
t.Initialize(); // Must call this to transfer values.
string resultText = t.TransformText();

//creates the code file if the name specified by the user has valid arguments
try
{
    System.IO.File.WriteAllText(System.IO.Path.Combine(path, fileNameTB.Text), resultText);
}
catch (ArgumentException exception)
{
    MessageBox.Show("That file name has invalid arguments!");
    return;
}

MessageBox.Show("Code Generated!");

Process.Start(path); // using System.Diagnostics -> opens the folder with generated files
```

Figura 3.8: Envio de dados para o *template*, utilizando *C#*

As primeiras instruções da figura 3.8 permitem definir o diretório onde o ficheiro gerado é guardado e caso este não exista criá-lo. Nas instruções seguintes, os dados são enviados para o *template*, que é executado utilizando as listas e variáveis que correspondem ao dados introduzidos pelo utilizador na *UI*. A pasta que contém o código gerado é aberta de forma automática, caso o nome do ficheiro atribuído pelo utilizador na aplicação não contenha caracteres inválidos.

### 3.2 Funcionalidades das máquinas de estados criadas na aplicação

Um fator determinante, que necessitava de ser definido na fase inicial do desenvolvimento do projeto, passou pela definição do conjunto de funcionalidades que as máquinas de estados especificadas pelo utilizador na aplicação seriam capazes de executar.

Todas as fases do projeto que viriam posteriormente, desde o desenvolvimento da *UI*, até à geração automática do código, dependeriam deste fator.

Como foi mencionado na secção 2.1, existem vários formalismos que permitem modelar uma máquina de estados. Cada um deles apresenta diferenças na forma de modelação, bem como no conjunto de possíveis funcionalidades que a máquina de estados pode executar.

O código gerado será aplicado em sistemas embebidos, por isso, para que a aplicação desenvolvida tivesse utilidade, as funcionalidades que a *UI* permite associar às máquinas de estados nela desenvolvidas deveriam depender diretamente dos comportamentos que um microprocessador/microcontrolador tem por hábito apresentar, tais como, eventos que podem ter origem em interrupções do sistema, na indicação do expirar de um *timer*, no valor de um *input* do sistema, assim como a execução de ações que permitam a alteração de valores de *outputs* do sistema, manipulação de variáveis ou execução de funções.

As funcionalidades escolhidas passam por uma solução híbrida das características de ambos os modelos de *Moore* e *Mealy*, da norma *UML* para especificação de *statecharts*, bem como do conceito de macro-ações presente na norma *GRAFCET*.

A aplicação desenvolvida permite especificar quatro tipos de ações: *transition actions*, que são executadas na transição entre estados, ou *entry actions*, *exit actions* e *continuous actions* que são associadas ao estados do sistema.

No que respeita às transições, permite especificar dois tipos: transições onde o estado destino é diferente do estado de partida e transições onde estes são iguais, *self-transitions*.

Na ocorrência de uma *self-transition*, foi definido que as ações associadas ao estado também são executadas porque, efetivamente, ocorre uma saída do estado atual e uma entrada neste mesmo. Sendo assim, no disparo deste tipo de transição, caso existam, são executadas as *transition*, *on exit* e *on entry actions*.

Também é permitida a existência de mais do que uma máquina a correr simultaneamente no projeto, utilizando o conceito de ortogonalidade onde é permitida a existência de duas ou mais regiões ortogonais, independentes e concorrentemente ativas. Através das funcionalidades hierárquicas que serão discutidas na secção seguinte, essas máquinas podem sincronizar os seus comportamentos e efetuar ações que alterem o comportamento umas das outras.

### 3.3 Funcionalidades do código gerado pela aplicação

O código que a aplicação gera pode assumir as linguagens *Arduino* (*C++*) ou *C*, podendo também o utilizador selecionar qual o método de estruturação da máquina de estados que pretende, métodos esses que podem assumir qualquer um dos formalismos mencionados na secção 2.4.

No caso da escolha da implementação em *Arduino*, a *UI* permite a associação de funções às ações que ativem/desativem pinos de saída, bem como de condições de transição que dependam de eventos temporais ou do valor de um pino de entrada ser "*HIGH*", "*LOW*", ocorrer o seu "*rising edge*" ou "*falling edge*". Todas essas funções podem ser associadas na *UI* de forma intuitiva e sem a necessidade de introduzir código manualmente, o gerador de código utiliza as funções da biblioteca *Arduino*, tais como o *digitalWrite()*, *digitalRead()*, *pinMode()* e *millis()* para executar tais funcionalidades.

No caso da implementação num microcontrolador/microprocessador que suporte a linguagem *C*, a aplicação deixa de permitir a associação das funcionalidades de geração automática de funções temporais e baseadas em pinos, já que, dependendo do modelo do sistema embebido alvo, essas instruções de código serão diferentes. Sendo assim, é permitida a associação desse tipo de funções caso o utilizador selecione a opção "*Code( )*" na qual pode introduzir manualmente as instruções corretas consoante o microprocessador/ microcontrolador que esteja a utilizar. Esta opção de introdução manual de código também pode ser associada a ações e eventos do *Arduino*.

Outra funcionalidade que se encontra disponível é a associação de ações hierárquicas, baseadas nas macro-ações da norma *GRAFCET* que permitem congelar, descongelar, desativar ou forçar a ativação de um estado de outra máquina de estados que se encontre noutra região ortogonal do projeto aberto na aplicação.

Na figura 3.9 pode ser visto um diagrama que resume as possibilidades de estruturação do código, bem como as linguagens de programação que podem ser geradas e as funções que podem ser associadas às ações e transições consoante a linguagem escolhida.

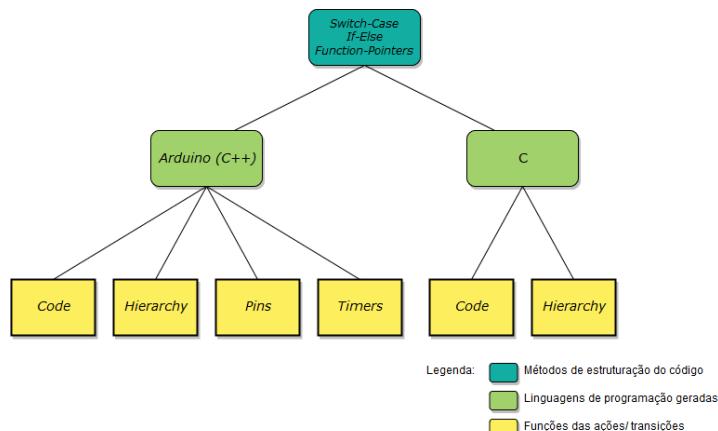


Figura 3.9: Diagrama das funcionalidades associadas ao código gerado na aplicação

### 3.4 User Interface

A *UI* tem como função permitir que o utilizador que nela interaja possa especificar todas as funcionalidades mencionadas nas secções 3.2 e 3.3, bem como conceder a possibilidade de obtenção do código correspondente, através de um "click" num botão desenhado para essa função.

Na figura 3.10 pode ser visto o ambiente de desenvolvimento para produzir a *UI* da aplicação.

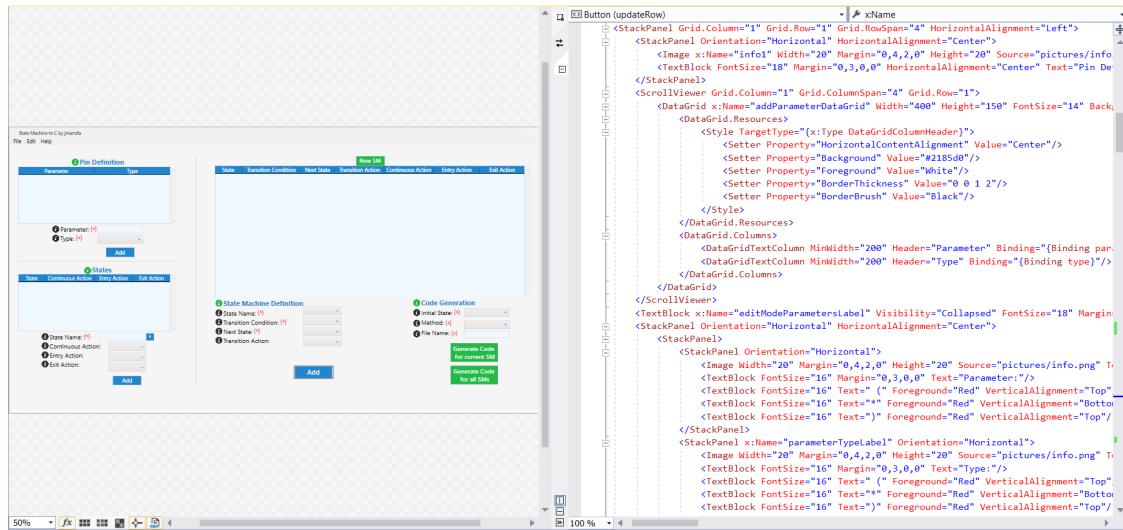


Figura 3.10: Desenvolvimento da camada *front-end* da aplicação no WPF

Toda a lógica que a ferramenta deve efetuar, desde funções executadas pelos botões, alterações das tabelas, surgimento de avisos ou mensagens de erro, funcionalidades de salvar/abrir projetos desenvolvidos na aplicação ou o envio de dados para os *T4 Text Templates* foi implementada na sua camada *back-end*, em C#, podendo uma porção desse código ser visto na figura seguinte.

The screenshot shows the Windows Presentation Foundation (WPF) development environment with a code editor window titled 'WpfApp4.MainWindow'. The code is written in C# and contains numerous event handlers for various UI controls. These include methods like 'AddState\_Click', 'EditState\_Click', 'DeleteState\_Click', 'UpdateStates\_Click', 'QuickStateName\_Click', and several 'GenerateCode\_Click' methods for different components. The code also includes logic for validating state names and checking if they are already in use. The code is heavily annotated with comments explaining its purpose.

```

private void AddState_Click(object sender, RoutedEventArgs e) //fill the table and check for errors ...
public int AddStateExceptions() //check for errors ...
private void AddStateDataGrid_SelectionChanged(object sender, SelectionChangedEventArgs e) //event executes if selection of row changes ...
private void EditState_Click(object sender, RoutedEventArgs e) //event executes after the user clicks on edit this row ...
private void DeleteState_Click(object sender, RoutedEventArgs e) //event executes after the user clicks on delete this row ...
private void UpdateStates_Click(object sender, RoutedEventArgs e) //event executes after the user clicks on update this row ...
public bool stateAlreadyExists(string s) //check if the state name is already being used ...
public bool stateIsBeingUsed(string stateNameToBeDeleted) //Verify if the deleted state is being used on State Machine Definition Table...
private void QuickStateName_Click(object sender, RoutedEventArgs e) //generate a default state name ...
//GENERATE CODE EVENT for selected SM, sends all the info to the TextTemplatePreProcessor File
public void GenerateCode_Click(object sender, RoutedEventArgs e){...}
//GENERATE CODE EVENT for all SMS of the project in a single file, sends all the info to the TextTemplatePreProcessor File
private void GenerateAllCode_Click(object sender, RoutedEventArgs e){...}
private void ComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e){...}
//Return to initial page
private void InitialPageSelector_Click(object sender, RoutedEventArgs e){...}
//Transition Events that are executed when combobox values change
private void TransitionComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e){...}
private void ComboBoxTransitionConditionH1_SelectionChanged(object sender, SelectionChangedEventArgs e){...}
//Out Events that are executed when combobox values change
private void OutGeneralComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e){...}

```

Figura 3.11: Desenvolvimento da camada *back-end* da aplicação no WPF

### 3.4.1 Manual de Utilização

Após a abertura da aplicação o utilizador efetua a escolha entre uma modelação de máquinas de estados para *Arduino* ou para um microcontrolador/microprocessador que utilize *C*. Sendo que, tal como mencionado na secção 3.3, a seleção da opção *Arduino*, implica o acesso a funções pré-definidas para ações, eventos temporais, ou funcionalidades baseadas em pinos do *Arduino*, enquanto que na seleção da opção de *C* estas funcionalidades podem ser implementadas mas introduzindo manualmente na *UI* as instruções de código que as efetuam, de acordo com o modelo do sistema embebido que estiver a utilizar.

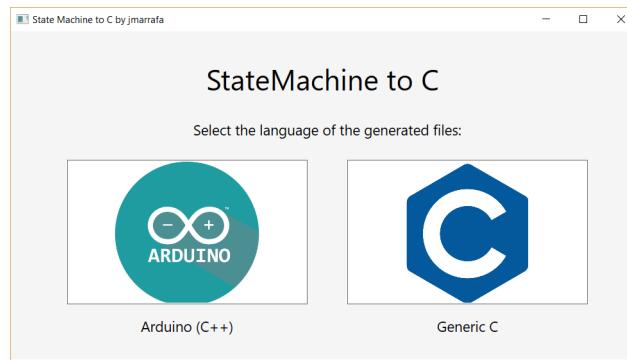


Figura 3.12: *Interface* de escolha da linguagem dos ficheiros gerados

Após a escolha da linguagem, o utilizador pode efetuar a modelação das máquinas de estado através de uma especificação tabelar. A diferença entre a escolha da opção de *C* em detrimento da opção *Arduino* (*C++*) implica apenas que a tabela de especificação dos pinos seja colapsada, sendo as restantes duas tabelas comuns às duas escolhas e consistem na de especificação dos estados e na de especificação das transições entre estados das máquinas de estados.

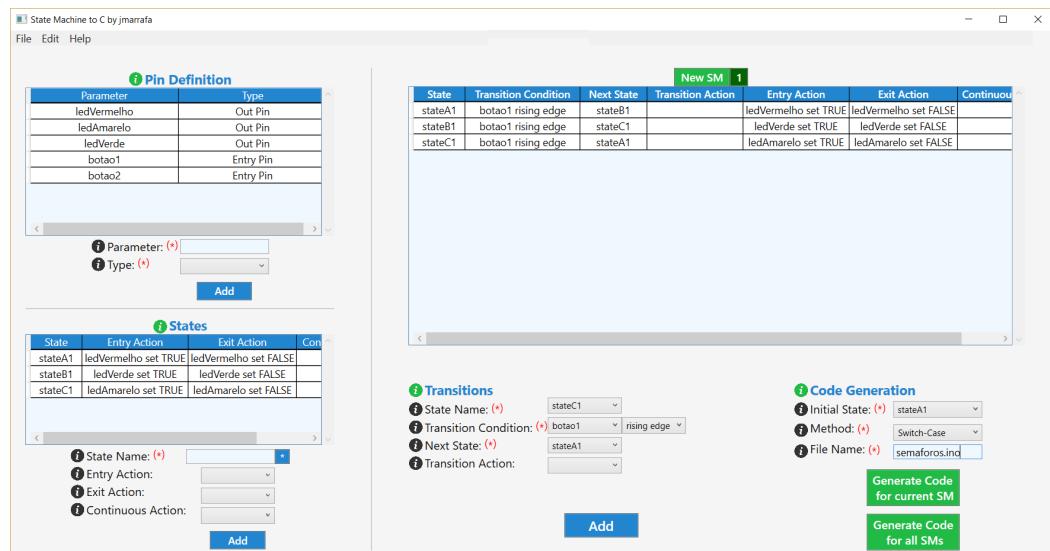


Figura 3.13: *Interface* da aplicação no caso da escolha *Arduino*

A primeira tabela, que só existe mediante a escolha *Arduino*, é a de especificação dos pinos e permite definir os parâmetros que irão corresponder a pinos do *Arduino*, de forma a que estes possam ser associados a funções pré-definidas para ações e transições, que têm como finalidade a geração automática de código de instruções da biblioteca *Arduino* (*digitalRead*, *digitalWrite*, *millis*, *loop*, *setup*, etc.) sem que o utilizador tenha de introduzir manualmente o código que efetua essas funções. Uma vez inseridos na tabela, esses parâmetros passam a poder ser selecionados nas *comboboxes* de todos os tipos de ações e eventos que provocam as transições de estados.

| Pin Definition |           |
|----------------|-----------|
| Parameter      | Type      |
| ledVermelho    | Out Pin   |
| ledAmarelo     | Out Pin   |
| ledVerde       | Out Pin   |
| botao1         | Entry Pin |
| botao2         | Entry Pin |

i Parameter: (\*) botao2  
 i Type: (\*) Entry Pin

Figura 3.14: Tabela de definição de pinos

A segunda tabela permite explicitar quais os estados que constituem a máquina de estados selecionada, bem como os 3 tipos de ação que esse estado pode ter a si associado. No exemplo da figura seguinte, verifica-se que, de facto, os parâmetros explicitados na primeira tabela podem ser associados a ações de ativação ou desativação do respetivo pino. Uma vez inserido um estado nesta tabela, este pode ser selecionado nas *comboboxes* para especificação de transições na terceira tabela, a de definição das transições.

| States  |                      |                       |       |
|---------|----------------------|-----------------------|-------|
| State   | Entry Action         | Exit Action           | Conti |
| stateA1 | ledVermelho set TRUE | ledVermelho set FALSE |       |
| stateB1 | ledVerde set TRUE    | ledVerde set FALSE    |       |
| stateC1 | ledAmarelo set TRUE  | ledAmarelo set FALSE  |       |

i State Name: (\*) stateC1 \*  
 i Entry Action: ledAmarelo set TRUE  
 i Exit Action: ledAmarelo set FALSE  
 i Continuous Action:

Figura 3.15: Tabela de definição dos estados e das suas ações

A terceira tabela completa a modelação da máquina de estados e permite explicitar as suas transições: qual o estado de partida e chegada, quais os eventos que as provocam e, caso existam, quais ações a si associadas. Uma vez definida, a máquina de estados encontra-se modelada e é possível obter o seu código equivalente.

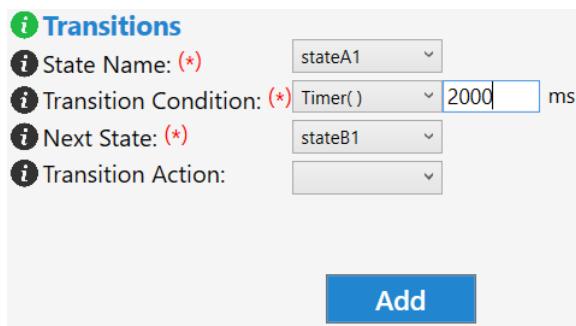


Figura 3.16: Associação de um evento temporal a uma transição

| New SM 1 2 3 4 X |                      |            |                         |                      |                       |                   |
|------------------|----------------------|------------|-------------------------|----------------------|-----------------------|-------------------|
| State            | Transition Condition | Next State | Transition Action       | Entry Action         | Exit Action           | Continuous Action |
| stateA1          | 2000ms               | stateB1    |                         | ledVermelho set TRUE | ledVermelho set FALSE |                   |
| stateB1          | botao1 rising edge   | stateC1    |                         |                      |                       | contador++;       |
| stateC1          | SM3-is on stateB3    | stateA1    | SM2-Go to State-stateC2 |                      |                       |                   |

Figura 3.17: Tabela de definição das transições e das suas ações

A aplicação e o gerador de código foram desenvolvidos de forma a suportarem o conceito de regiões ortogonais, nas quais são permitidas a existência de mais do que uma máquina de estados a correr paralelamente. Para tal é utilizado o menu que se encontra por cima da tabela de transições da figura anterior, que permite selecionar, apagar ou criar uma nova máquina de estados. O utilizador deve estar ciente que no caso de eliminar uma máquina de estados, tal irá implicar a remoção automática de todas as ações e eventos hierárquicos, associados à máquina eliminada, declarados nas outras máquinas.

Os quatro tipos de ações (*on entry*, *on exit*, *on transition*, *continuous*) que podem ser associados à máquina podem ter três tipos de funções (*Code*, *Hierarchy* e *Pin Activation/Deactivation*):

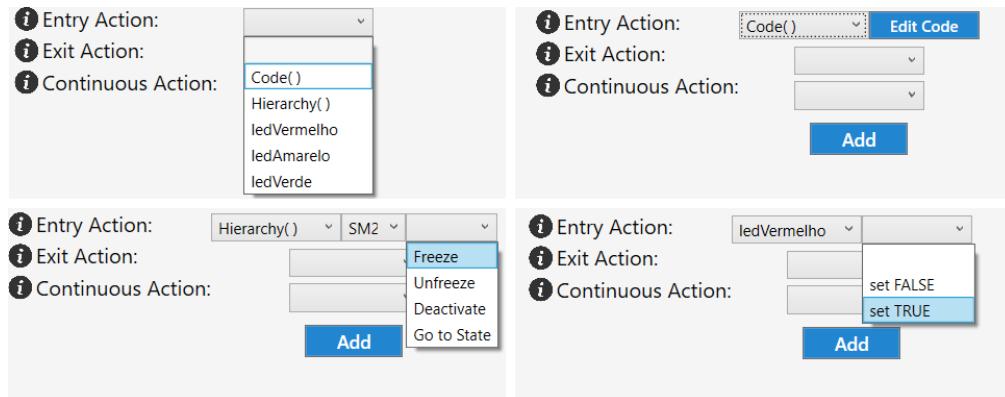


Figura 3.18: Conjunto de funcionalidades que podem ser associadas às ações

Na figura 3.18 podem ser vistas as alterações na estrutura da *UI* mediante as seleções das diferentes opções de função para a ação.

A opção *Hierarchy* permite efetuar o congelamento, descongelamento, desativação e forçagem de um estado de outra máquina que exista noutra região ortogonal e encontra-se presente, quer na escolha de geração de linguagem para *Arduino(C++)*, quer na escolha de *C*.

A opção *Pin Activation/Deactivation* permite efetuar a função de ativação/ desativação de um parâmetro definido na tabela de definição de pinos e só é possível na escolha da opção *Arduino(C++)*, gerando automaticamente no ficheiro de texto as funções como o *digitalWrite()*, *digitalRead()* e *setup()* que existem na biblioteca *Arduino*.

A opção *Code* permite inserir instruções manualmente e encontra-se presente, quer na escolha de geração de linguagem para *Arduino(C++)*, quer na escolha de *C*. A sua existência permite associar qualquer tipo de instrução à ação/transição, possibilitando no caso de *C*, que não tem acesso à funcionalidade *Pin Activation/Deactivation*, introduzir, por exemplo, a seguinte instrução:

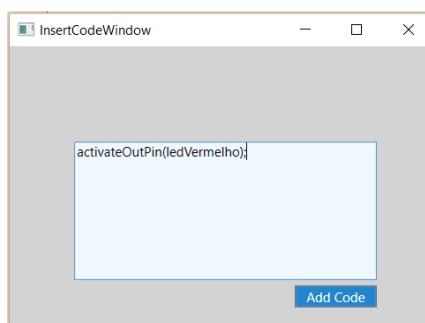


Figura 3.19: Interface que permite a introdução de instruções manualmente

A instrução definida ficará associada à ação e estará presente no código gerado, sendo que o utilizador necessita apenas de incluir no ficheiro gerado a definição da função de acordo com a forma correta de programar o seu microprocessador/microcontrolador.

As transições que podem ser associadas à máquina podem ter quatro tipos de funções (*Code*, *Timer*, *Hierarchy*, *Pin Value*):

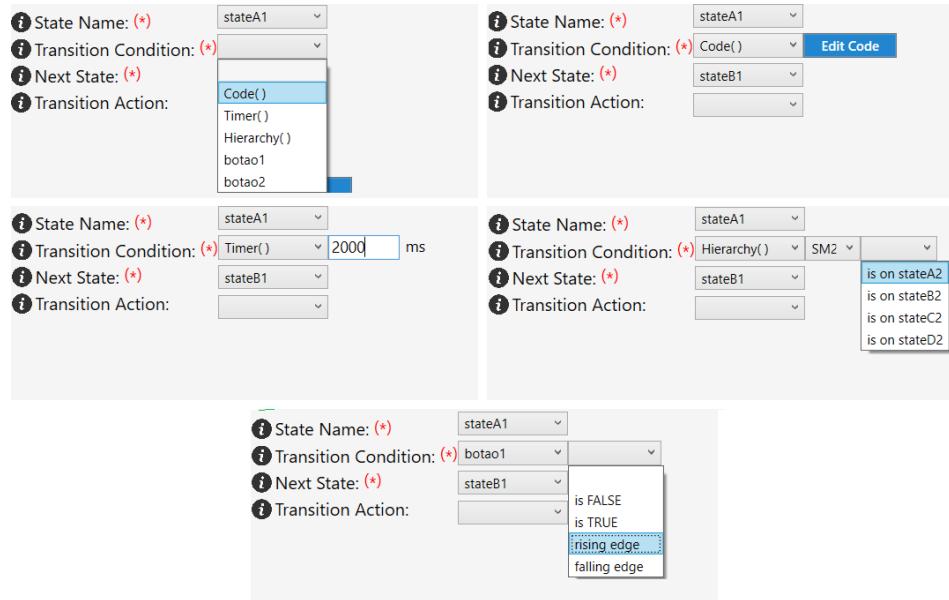


Figura 3.20: Conjunto de funcionalidades que podem ser associadas às transições

Na figura 3.20 podem ser vistas as alterações na estrutura da *UI* mediante as seleções das diferentes condições de disparo da transição.

A opção *Hierarchy* permite sincronizar os comportamentos das várias máquinas através do disparo da transição consoante o estado ativo de outra máquina que exista noutra região ortogonal e encontra-se presente, quer na escolha de geração de linguagem para *Arduino*(*C++*), quer na escolha de *C*.

As opções *Timer* e *Pin Value* só se encontram pré-definidas e disponíveis para a opção *Arduino*, mas mais uma vez podem ser implementadas para a opção de *C*, utilizando a função *Code*, introduzindo a instrução manualmente.

Uma vez especificadas as máquinas de estados nas respetivas tabelas, o ficheiro que a descreve em código pode ser obtido preenchendo os campos relativos à geração do código e efetuando o "click" numa das duas opções de geração de código disponibilizadas.

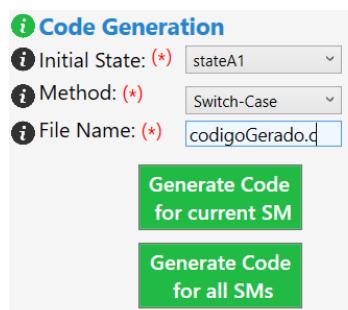


Figura 3.21: Interface da secção para geração do ficheiro de código

A primeira opção permite gerar código correspondente apenas à máquina de estados que se encontre, no momento, selecionada no projeto.

O segundo gera código correspondente a todas as máquinas existentes no projeto, que são codificadas de forma a correrem paralelamente.

Caso a máquina selecionada contenha ações ou eventos hierárquicos, o "click" no primeiro botão irá gerar um aviso de erro, já que a existência desse tipo de ações/eventos implica a existência de mais do que uma máquina.

O campo de especificação do estado inicial deve ser preenchido em todas as máquinas de estados do projeto. Os outros dois parâmetros são comuns a todas: o método de geração escolhido pode assumir os formalismos vistos na secção 2.4 e o campo "File Name" permite especificar o nome ficheiro gerado.

De forma a facilitar a interpretação de cada campo de preenchimento, foram adicionados ícones de informação associados a cada um destes, que permitem aceder a uma breve descrição da função desse mesmo campo.

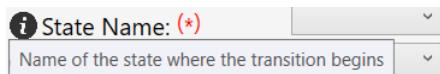


Figura 3.22: Ícones de informação

Qualquer tentativa de adição de comportamentos que não respeitem as normas de modelação de uma máquina de estados, tais como, a tentativa de introdução de um estado de partida com uma transição, mas sem estado de chegada e vice-versa ou a repetição de condições de transição para múltiplas transições que partem de um mesmo estado, originam mensagens de erro. Tal funcionalidade faz com que a aplicação não só permita gerar código, mas também efetue a verificação do cumprimento das normas e não permita a introdução de comportamentos errados.

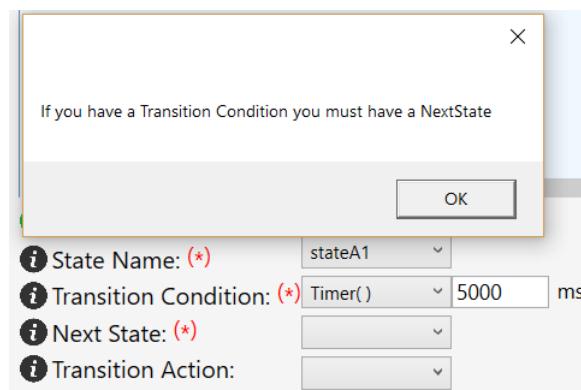


Figura 3.23: Exemplo de *MensajeBox* de erro no não cumprimento de normas de modelação

Com o intuito de permitir a reutilização do programa, foi implementada a funcionalidade de gravação e abertura de projetos criados na aplicação.

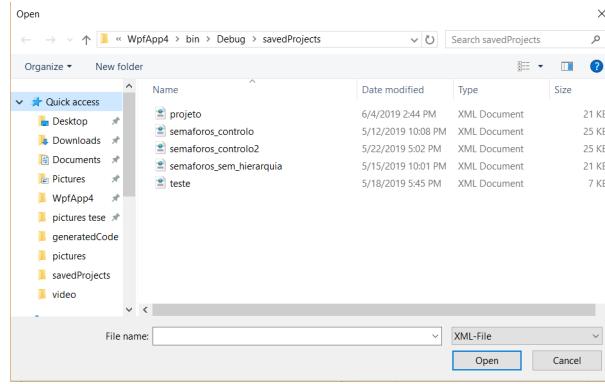


Figura 3.24: Interface da abertura de um *OpenFileDialog*

A gravação de um projeto é realizada através da serialização de um objeto da classe "savedProject", que contém todas as listas de dados do projeto aberto, para um ficheiro XML.

A abertura de um projeto consiste na desserialização do ficheiro XML que contém o projeto gravado e na introdução dos seus dados nas respetivas listas, seguido da invocação do método "loadProjectParameters" que, com base nessas listas, preenche todas as tabelas e campos que se encontravam preenchidos no momento da gravação do ficheiro.

```
<?xml version="1.0" encoding="utf-8"?>
<savedProject xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
|   <stateMachineList>
|       <stateMachine>
|           <allStatesList>
|               <string>stateA1</string>
|               <string>stateB1</string>
|               <string>stateC1</string>
|           </allStatesList>
|           <tableParametersList>
|               <tableParameters>
|                   <stateName>stateA1</stateName>
|                   <stateNameToTable>stateA1</stateNameToTable>
|                   <transition><Timer( )></transition>
|                   <transitionToTable>5000ms</transitionToTable>
|                   <nextStateName>stateB1</nextStateName>
|                   <nextStateNameToTable>stateB1</nextStateNameToTable>
|                   <outOnTrans />
|                   <outOnContinue />
|                   <outOnEntry>ledVermelho</outOnEntry>
|                   <outOnEntryToTable>ledVermelho set TRUE</outOnEntryToTable>
|                   <outOnExit>ledVermelho</outOnExit>
|                   <outOnExitToTable>ledVermelho set FALSE</outOnExitToTable>
|                   <transitionType>Timer( )</transitionType>
|                   <transitionValue>5000</transitionValue>
|                   <outOnEntryType>Out Pin</outOnEntryType>
|                   <outOnEntryValue>set TRUE</outOnEntryValue>
|                   <outOnExitType>Out Pin</outOnExitType>
|                   <outOnExitValue>set FALSE</outOnExitValue>
|               </tableParameters>
|               <tableParameters>
|                   <stateName>stateB1</stateName>
|                   <stateNameToTable>stateB1</stateNameToTable>
|                   <transition><Timer( )></transition>
|                   <transitionToTable>5000ms</transitionToTable>
|                   <nextStateName>stateC1</nextStateName>
|                   <nextStateNameToTable>stateC1</nextStateNameToTable>
|                   <outOnTrans />
|                   <outOnContinue />
|                   <outOnEntry>ledVerde</outOnEntry>
|                   <outOnEntryToTable>ledVerde set TRUE</outOnEntryToTable>
|                   <outOnExit>ledVerde</outOnExit>
|                   <outOnExitToTable>ledVerde set FALSE</outOnExitToTable>
|                   <transitionType>Timer( )</transitionType>
|                   <transitionValue>5000</transitionValue>
|                   <outOnEntryType>Out Pin</outOnEntryType>
|                   <outOnEntryValue>set TRUE</outOnEntryValue>
|                   <outOnExitType>Out Pin</outOnExitType>
|                   <outOnExitValue>set FALSE</outOnExitValue>
```

Figura 3.25: Ficheiro XML de um projeto gravado na aplicação

Os nomes dos estados podem ser explicitados manualmente:

|                             |                                  |   |
|-----------------------------|----------------------------------|---|
| <b>i State Name: (*)</b>    | activateLedState                 | * |
| <b>i Entry Action:</b>      | <input type="button" value="▼"/> |   |
| <b>i Exit Action:</b>       | <input type="button" value="▼"/> |   |
| <b>i Continuous Action:</b> | <input type="button" value="▼"/> |   |

Figura 3.26: Atribuição manual de um nome a um estado

No entanto, de forma a acelerar a fase de especificação do seu nome e tendo em conta que o nome que lhe for atribuído em nada afeta o funcionamento do programa, é possível gerá-lo de forma automática, através do "click" no botão "\*", gerando-o segundo uma notação onde a *string "state"* é seguida de um caractere alfabético variável, que por sua vez é seguido por um caractere numérico que corresponde ao número da máquina à qual pertence.

|                             |                                  |   |
|-----------------------------|----------------------------------|---|
| <b>i State Name: (*)</b>    | stateA1                          | * |
| <b>i Entry Action:</b>      | <input type="button" value="▼"/> |   |
| <b>i Exit Action:</b>       | <input type="button" value="▼"/> |   |
| <b>i Continuous Action:</b> | <input type="button" value="▼"/> |   |

Figura 3.27: Atribuição automática de um nome a um estado

Qualquer uma das tabelas pode ser alterada através da seleção da linha com os parâmetros que se pretende alterar:

|           |           |
|-----------|-----------|
| ledVerde2 | Out Pin   |
| botao1    | Entry Pin |
| botao2    | Entry Pin |

|                         |                                  |
|-------------------------|----------------------------------|
| <b>i Parameter: (*)</b> | <input type="text"/>             |
| <b>i Type: (*)</b>      | <input type="button" value="▼"/> |
| <b>Edit</b>             | <b>Delete</b>                    |
| <b>Add</b>              |                                  |

Figura 3.28: Atribuição automática de um nome a um estado

## 3.5 Código gerado

Tendo as funcionalidades que as máquinas de estados podem executar definidas e desenvolvida a *UI* que permite a sua especificação, procedeu-se à codificação dos *T4 text templates* que geram os ficheiros de texto que contêm o código correspondente às máquinas de estados especificadas nas linguagens *Arduino(C++)* ou *C*.

A forma de estruturação da máquina de estados pode assumir os formalismos "*switch-case*", "*if-else*" ou apontadores para funções, consoante a especificação do utilizador na *UI*.

As ações podem estar associadas a um estado, ocorrendo à sua entrada, saída ou continuamente, ou podem ser associadas a uma transição, podendo assumir funções pré-definidas, sendo estas: código inserido diretamente pelo utilizador, ativação e desativação de pinos ou ações hierárquicas. No caso dos eventos, estes podem assumir funções de código inserido pelo utilizador, de disparo consoante valores dos pinos, eventos hierárquicos ou eventos temporais.

Sendo assim, os ficheiros *T4 Text Templates* desenvolvidos têm como função percorrer as listas de dados que lhes são enviadas após o "click" do botão "*Generate Code*" na *UI* e colocar os seus dados nas secções do ficheiro de código gerado apropriadas, de forma a que este corresponda, de facto, ao comportamento correto do sistema.

Foram criados quatro *templates*:

- Um efetua a geração do código em *Arduino (C++)*, em qualquer formalismo que o utilizador especifique, considerando apenas a máquina selecionada na aplicação;
- Um efetua a geração do código em *Arduino (C++)*, em qualquer formalismo que o utilizador especifique, considerando todas as máquinas do projeto;
- Um efetua a geração dos ficheiros *header* e *source* em *C*, em qualquer formalismo que o utilizador especifique, considerando apenas a máquina selecionada na aplicação;
- Um efetua a geração dos ficheiros *header* e *source* em *C*, em qualquer formalismo que o utilizador especifique, considerando todas as máquinas do projeto.

O código desenvolvido em cada um destes ficheiros consiste numa sequência vasta de instruções *for* que englobam outras instruções do mesmo tipo que, por sua vez, percorrem listas que contêm outras listas, seguidas de instruções condicionais *if-else* que, consoante os tipos de ações ou eventos, bem como tendo em conta o número de máquinas e de estados que as constituem, imprimem diferentes instruções de código nas respetivas secções corretas do ficheiro.

Uma parte significativa do desenvolvimento deste projeto consistiu na codificação destes *templates*. No entanto, a descrição exaustiva desse código não produziria valor para dissertação. Por este motivo, serão mostrados apenas alguns excertos do código, codificado em C#, bem como o código que foi gerado como consequência.

Tomando como exemplo o *template* que gera o ficheiro na linguagem *Arduino C++* para mais do que uma máquina de estados, a seguinte instrução da figura 3.29 implica que seja gerada, no ficheiro de texto, uma instrução *switch* para cada máquina de estados modelada pelo utilizador na

*UI*, bem como que para cada estado que lhe pertença seja impresso um *case* que descreva o seu comportamento.

```
void runSMs()
{
    <#for(int z=0; z<stateMachineList.Count; z++){#>

    //State Machine <#=z+1#>
    switch(currentState<#=z+1#>)
    {
        <#foreach(string item in stateMachineList[z].allStatesList){#>

        case <#=item#>:
```

Figura 3.29: Secção de código para geração de um *switch* para cada máquina, utilizando *T4*

```
void runSMs()
{
    //State Machine 1
    switch(currentState1)
    {
        case stateA1:
            | |
            break;

        case stateB1:
            | |
            break;

        default: break;
    }

    //State Machine 2
    switch(currentState2)
    {
        case stateA2:
            | |
            break;

        case stateB2:
            | |
            break;

        default: break;
    }
}
```

Figura 3.30: Código gerado como consequência das instruções da figura anterior

O código que sucede as instruções representadas na figura 3.29, verifica a existência, o valor e os tipos das ações associadas aos estados ou às transições, bem como os tipos e valores de eventos que disparem as transições. Consoante esses parâmetros, efetua a sua impressão no ficheiro de texto gerado.

Tomando como exemplo o *template* que gera o ficheiro na linguagem *Arduino C++* para apenas a máquina de estados selecionada, a seguinte instrução implica que seja gerada uma função para cada estado que o utilizador crie na *UI*, o que corresponde à implementação orientada a apontadores para funções.

```
<#foreach(string item in allStatesList){#>

void run<#=item#>()
{
    /* DEALS WITH ENTRY ACTIONS */
    <#for(int i=0; i<tableParametersList.Count; i++){ #>
        <if(tableParametersList[i].stateName == item){#>

            if(INIT==0)
            {
                previousMillis = currentMillis;
```

Figura 3.31: Secção de código para geração de um *switch* para cada máquina, utilizando *T4*

```

void runstateA0()
{
    if(INIT==0)
    {
        previousMillis = currentMillis;
        digitalWrite(ledVermelho, HIGH); //on entry action
        INIT=1;
    }

    timerState = currentMillis - previousMillis;

    if(digitalRead(botao1) == HIGH) //transition condition
    {
        currentState = stateB1;
        digitalWrite(ledVermelho, LOW); //exit action
        INIT = 0; //resets the init indicator
    }
}

void runstateB0()
{
    if(INIT==0)
    {
        previousMillis = currentMillis;
        digitalWrite(ledVermelho, HIGH); //on entry action
        INIT=1;
    }
}

```

Figura 3.32: Código gerado como consequência das instruções da figura anterior

A seguinte instrução permite gerar o código que executa uma ação à entrada de um estado. Para tal é efetuada uma verificação do tipo e valor da ação, imprimindo o seu código correspondente.

```

if(INIT<#=z+1#> == 0)
{
    previousMillis<#=z+1#> = currentMillis;
    <#if(stateMachineList[z].tableParametersList[i].outOnEntryType == "Code( )"){#>
        <#=stateMachineList[z].tableParametersList[i].outOnEntryValue#> //on entry action
        <# } #>
        <#if(stateMachineList[z].tableParametersList[i].outOnEntryType == "Out Pin"){#>
            <#if(stateMachineList[z].tableParametersList[i].outOnEntryValue == "set TRUE"){#>
                digitalWrite(<#=stateMachineList[z].tableParametersList[i].outOnEntry#>, HIGH); //on entry action
                <# } #>
                <#if(stateMachineList[z].tableParametersList[i].outOnEntryValue == "set FALSE"){#>
                    digitalWrite(<#=stateMachineList[z].tableParametersList[i].outOnEntry#>, LOW); //on entry action
                    <# } #>
                    <#if(stateMachineList[z].tableParametersList[i].outOnEntryType == "Hierarchy( )"){#>
                        <#if(getSecondPart(stateMachineList[z].tableParametersList[i].outOnEntryValue)=="Deactivate"){#>
                            deactivateSM<#=getFirstPart(stateMachineList[z].tableParametersList[i].outOnEntryValue)#> = true; //on entry action
                            <# } #>
                            <#if(getSecondPart(stateMachineList[z].tableParametersList[i].outOnEntryValue)=="Freeze"){#>
                                freezeSM<#=getFirstPart(stateMachineList[z].tableParametersList[i].outOnEntryValue)#> = true; //on entry action
                                <# } #>
                                <#if(getSecondPart(stateMachineList[z].tableParametersList[i].outOnEntryValue)=="Unfreeze"){#>
                                    freezeSM<#=getFirstPart(stateMachineList[z].tableParametersList[i].outOnEntryValue)#> = false; //on entry action
                                    <# } #>
                                    <#if(getSecondPart(stateMachineList[z].tableParametersList[i].outOnEntryValue)=="Go to State"){#>
                                        <# string auxIndex = getFirstPart(stateMachineList[z].tableParametersList[i].outOnEntryValue); #>
                                        <# string auxState = getThirdPart(stateMachineList[z].tableParametersList[i].outOnEntryValue); #>

```

```

forcedCurrentState<#=auxIndex#> = <#=auxState#>;
forceStateSM<#=auxIndex#> = true;
<# } #>
<# } #>
INIT<#=z+1#>=1;
}

```

Figura 3.33: Secção de código para geração de ações à entrada de um estado, utilizando T4

```

case stateA1:
    if(INIT1 == 0)
    {
        previousMillis1 = currentMillis;
        digitalWrite(ledVermelho, HIGH); //on entry action
        INIT1=1;
    }
}

```

Figura 3.34: Código gerado como consequência das instruções da figura anterior

A ação é efetuada apenas uma vez, quando a variável auxiliar *INIT1*, cujo caractere numérico indica a máquina de estados a que pertence, tem o valor 0. Uma vez executada, esse valor passa a 1, só voltando a assumir o valor 0 na ocorrência de uma transição para o estado seguinte.

A seguinte instrução representa uma amostra do código que verifica o tipo de transição e imprime a expressão condicional que provoca o disparo de uma transição entre estados.

```

<#for(int i=0; i<stateMachineList[z].tableParametersList.Count; i++){ #>
    <if(stateMachineList[z].tableParametersList[i].stateName == item){#>
        <if((stateMachineList[z].tableParametersList[i].transition != "")&&
(stateMachineList[z].tableParametersList[i].transitionType == "Code( )")){#>

            if((#=stateMachineList[z].tableParametersList[i].transitionValue#)<if(stateMachineList[z].canBeFrozen){#> &&
(!freezeSM<#=z+1#>)<# } #>
                <# } #
                <if((stateMachineList[z].tableParametersList[i].transition != "")&&
(stateMachineList[z].tableParametersList[i].transitionType == "Timer( )")){#>

                    if ((timerState<#=z+1#> >= timer<#=(char)timerAux#><#=z+1#>)<if(stateMachineList[z].canBeFrozen){#> &&
(!freezeSM<#=z+1#>)<# } #>) //transition condition
                        <# timerAux++;#
                    <# } #
                    <if((stateMachineList[z].tableParametersList[i].transition != "")&&
(stateMachineList[z].tableParametersList[i].transitionType == "Hierarchy( )")){#>

```

Figura 3.35: Secção de código para geração de condições de transição, utilizando T4

```

case stateA1:
    if(INIT1 == 0)
    {
        previousMillis1 = currentMillis;
        digitalWrite(ledVermelho, HIGH); //on entry action
        INIT1=1;
    }

    timerState1 = currentMillis - previousMillis1;

    if ((timerState1 >= timerA1) && (!freezeSM1)) //transition condition
    {
        currentState1 = stateB1;
        digitalWrite(ledVermelho, LOW); //exit action
        INIT1 = 0; //resets the init indicator
    }
}

```

Figura 3.36: Código gerado como consequência das instruções da figura anterior

No caso da figura 3.36, o utilizador especificou na *UI* que na primeira máquina de estados, o estado A transita para o estado B na ocorrência de um evento temporal, tendo o estado A a si associado uma ação à saída de desativação do pino que corresponde ao parâmetro "ledVermelho".

A principal diferença na estrutura dos ficheiros gerados entre a escolha de uma geração para a linguagem *Arduino* (*C++*) ou *C* passa pelo facto de, no caso do *C*, não ser gerado apenas um ficheiro que possa ser diretamente implementável no microcontrolador/microprocessador alvo, mas dois ficheiros: um *header* e uma *source*, que devem ser incluídos no projeto do utilizador e chamada a função de execução das máquinas de estados "*runSMs()*".

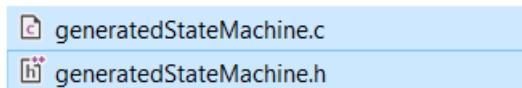


Figura 3.37: Ficheiros gerados na escolha da linguagem *C*

```
#include "generatedStateMachine.h"
```

Figura 3.38: Inclusão dos ficheiros gerados no projeto do utilizador

Uma possível implementação, por exemplo, para o microcontrolador TMS320F2803x seria após a inclusão, chamar a função através de uma interrupção, tal como pode ser visto na figura seguinte:

```
// This function will be called periodically with a
// period of CPUTIMER1_PERIOD_US micro-seconds.
#pragma CODE_SECTION(cpu_timer1_isr, "RamFunctions");
__interrupt void cpu_timer1_isr(void) {
    static u16 CntLed3 = 0;
    cpu_timer1_isr_GPIO_SET

    // The ControlCard Led3 will blink a every second.
    if (CntLed3 == 0) {
        ControlCard_Led3_TOGGLE
        CntLed3 = 499;
    }
    else {
        CntLed3 = CntLed3 - 1;
    }

    // Call the StateMachines procedures
    runSMs();
```

```
    cpu_timer1_isr_GPIO_CLEAR
    // The CPU acknowledges the interrupt.
```

```
}
```

Figura 3.39: Exemplo de uma implementação para o microcontrolador TMS320F2803x

### 3.5.1 Ações hierárquicas

A existência de ações e eventos hierárquicos é um dos fatores diferenciadores da aplicação criada em relação ao que já existe no mercado. Estas ações foram baseadas na norma *GRAFCET* e permitem congelar, descongelar, desativar ou forçar um estado de uma outra máquina de estados existente no projeto.

Nesta secção será explicada a lógica do código gerado quando o utilizador especifica na *UI* a utilização deste tipo de ações/eventos.

A associação de uma ação hiérarquica que altere o funcionamento de outra máquina de estados implica a geração, de forma automática, de inicializações de variáveis *booleanas* relacionadas com os tipos de ações e com a máquina alvo que estas irão afetar.

No caso do excerto de código gerado da figura 3.40, a impressão de tais instruções implica a existência de ações hierárquicas que irão afetar as máquinas 1 e 2.

```
bool deactivateSM1 = false;
bool freezeSM1 = false;
bool forceStateSM1 = false;

bool deactivateSM2 = false;
bool freezeSM2 = false;
bool forceStateSM2 = false;
```

Figura 3.40: Variáveis *booleanas* geradas para execução de ações hierárquicas

#### 3.5.1.1 Congelamento/ Descongelamento

A presença de uma ação de congelamento no projeto implica que, na secção de código onde tal ação é executada, seja atribuído o valor "true" à variável *booleana* da máquina que se pretende controlar. No caso da figura 3.41, tal ação é realizada à entrada do estado B3, mas podia ser executada em qualquer um dos restantes três tipos de ações.

```
case stateB3:

    if(INIT3 == 0)
    {
        previousMillis3 = currentMillis;
        freezeSM1 = true; //on entry action
        INIT3=1;
    }
```

Figura 3.41: Código gerado para execução de ações de congelamento

A existência deste tipo de ação, neste caso a ser efetuada pela máquina 3, implica que na máquina que será congelada, neste caso a máquina 1, todas as suas condições de transição entre estados serão complementadas com a instrução de esta "não estar congelada". Segundo a definição de que uma máquina de estados encontra-se congelada quando as suas transições não disparam, mesmo que o evento que as provoque ocorra.

```

case stateA1:
    if(INIT1 == 0)
    {
        previousMillis1 = currentMillis;
        digitalWrite(ledVermelho, HIGH); //on entry action
        INIT1=1;
    }

    timerState1 = currentMillis - previousMillis1;

    if ((timerState1 >= timerA1) && (!freezeSM1)) //transition condition
    {
        currentState1 = stateB1;
        digitalWrite(ledVermelho, LOW); //exit action
        INIT1 = 0; //resets the init indicator
    }
}

```

Figura 3.42: Alterações no código da máquina alvo de uma ação de congelamento

Já o seu descongelamento é efetuado através da alteração da variável *booleana* da máquina, novamente, para "false".

### 3.5.1.2 Desativação

No caso das ações de desativação de uma máquina de estados, a sua execução consiste, também, na atribuição do valor "true" à variável *booleana* da máquina que se pretende desativar. No entanto, as alterações na estrutura do código da máquina alvo são diferentes, pois se no caso do congelamento a adição da condição "não estar congelada" em todas as transições da máquina era suficiente, neste caso tal não é pretendido.

Sendo assim, foi seguida a definição de que a desativação de uma máquina de estados consiste na sua transição para o estado de desativação, no qual permanece sem efetuar qualquer tipo de ação até que uma ação de forçagem para um determinado estado ative novamente a máquina. Foi considerado que, mediante uma ação de desativação, na existência de ações de saída associadas ao estado em que a máquina alvo se encontre, essas ações sejam executadas antes de ocorrer transição para o estado de desativação.

A figura 3.43 representa a instrução que é adicionada a todos os estados que pertencem à máquina de estados alvo de uma ação de desativação, onde a alteração da variável *booleana* para "true" provoca não só a transição para o estado de desativação, mas também, caso existam, a execução das ações de saída do estado em que o sistema de encontre.

```

if(deactivateSM1 == true)
{
    currentState1 = SM1_DEACTIVATED;
    digitalWrite(ledVermelho, LOW); //exit action
    INIT1 = 0;
}

```

Figura 3.43: Alterações no código da máquina alvo de uma ação de desativação

A adição do estado de desativação é realizada através da inclusão de uma instrução "case" na máquina alvo. A única forma da máquina sair do estado de desativação é a aplicação de uma ação de forçagem de estado.

```
case SM1_DEACTIVATED:

    if(forceStateSM1 == true)
    {
        currentState1 = forcedCurrentState1;
        forceStateSM1 = false;
        deactivateSM1 = false;
    }
break;
```

Figura 3.44: Alterações no código da máquina alvo de uma ação de desativação

### 3.5.1.3 Forçagem de um estado

A execução de uma ação de forçagem de estado é efetuada não só através da ativação da sua variável *booleana*, mas também da associação do estado para o qual se pretende que a máquina alvo transite à variável "estado forçado".

```
if(INIT3 == 0)
{
    previousMillis3 = currentMillis;
    forcedCurrentState1 = stateA1;
    forceStateSM1 = true;
    INIT3=1;
}
```

Figura 3.45: Código gerado para execução de ações de forçagem de estado

Para além disso, é adicionada a seguinte instrução de código em todos os estados da máquina alvo, notando que, tal como no caso da desativação, as ações de saída, caso existam, são executadas antes da ocorrência de mudança de estado.

```
if(forceStateSM1 == true)
{
    currentState1 = forcedCurrentState1;
    forceStateSM1 = false;
    digitalWrite(ledAmarelo, LOW); //exit action
    INIT1 = 0;
}
```

Figura 3.46: Alterações no código da máquina alvo de uma ação de forçagem de estado

Note-se que, todo o código explicado neste capítulo é gerado de forma automática, através da escolha da opção "*Hierarchy( )*", disponível na fase de modelação da máquina de estados. Não havendo, por isso, a necessidade de o utilizador introduzir qualquer tipo de instrução manualmente para que as funcionalidades hierárquicas sejam executadas.

# Capítulo 4

## Testes e Resultados

Com o intuito de verificar se o gerador de código desenvolvido correspondia, efetivamente, aos modelos introduzidos pelo utilizador na *UI*, ao longo do desenvolvimento do projeto foram sempre efetuados testes a cada nova funcionalidade introduzida, utilizando para tal uma *breadboard*, constituída por *LEDs* que pretendiam representar a ativação e desativação de pinos de saída e por botões que permitiam alterar o valor dos pinos de entrada. Como alvo do código gerado foi utilizada uma placa *Arduino Uno*, que continha o microcontrolador ATMega328P. A sua programação foi efetuada através da utilização do *Arduino IDE*, tendo sido introduzidas apenas as instruções de código geradas pela aplicação. Utilizando uma comunicação UART foi possível imprimir no *PC* informações relevantes para a verificação do correto funcionamento do sistema, tal como a impressão de uma indicação na ocorrência de uma mudança de estados.

### 4.1 Problema 1 - Controlo de dois semáforos

#### 4.1.1 Descrição do Problema

A aplicação permite gerar código em *Arduino(C++)* ou em C. Tendo em conta que a escolha da opção *Arduino(C++)* implica o acesso a mais funcionalidades, tal como a possibilidade de associação de funções pré-definidas às ações e transições, que se baseiam em eventos temporais ou em parâmetros introduzidos na tabela de definição de pinos, o primeiro problema que foi desenvolvido para testar o correto funcionamento da aplicação foi modelado de forma a testar um grande número de funcionalidades que a escolha dessa opção disponibiliza.

As funcionalidades que foram testadas incluem: ações à entrada, à saída e à transição, assumindo funções de ativação e desativação de pinos, de código introduzido pelo utilizador e de execução de ações hierárquicas de congelamento, descongelamento, forçagem de estados e desativação de outras máquinas. Já no caso dos eventos, estes assumem funções com base em valores de pinos de entrada, no tempo decorrido e de verificação dos estados ativos de outras máquinas de estados. A existência de ações e eventos hierárquicos implica, portanto, que o problema contenha mais do que uma máquina de estados em diferentes regiões ortogonais. Na figura 4.1 encontram-se representadas, segundo a norma *UML*, as quatro máquinas que modelam este problema.

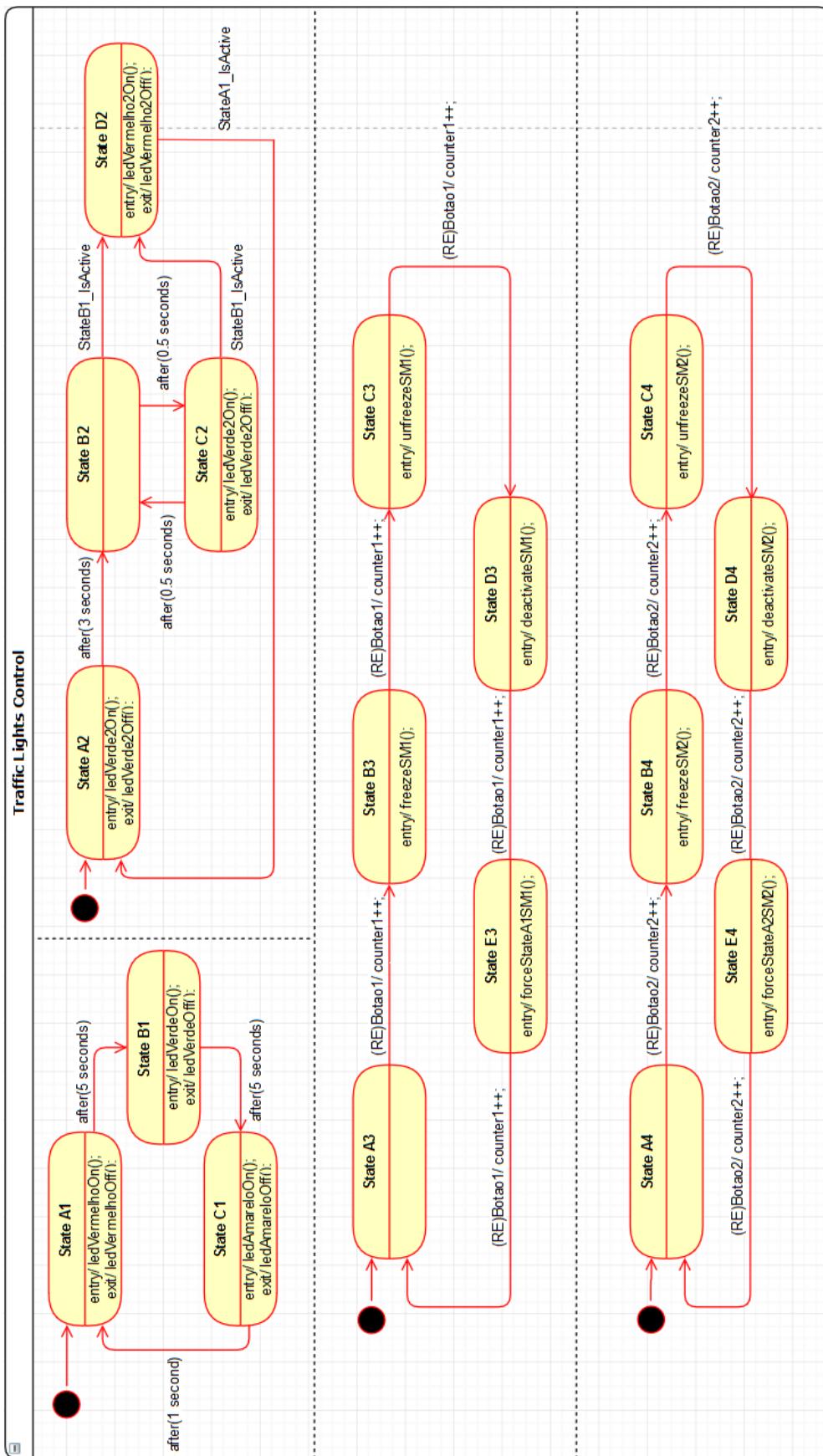


Figura 4.1: UML statechart para controlo de 2 semáforos

O problema consiste no controlo de dois semáforos através de quatro máquinas de estado:

- A primeira permite controlar um semáforo para veículos que pode transitar entre três estados mediante a ocorrência de eventos temporais, efetuando à entrada e à saída dos estados ações de ativação e desativação de pinos de saída;
- A segunda modela o comportamento de um semáforo para peões que pode transitar entre quatro estados, não só através de eventos temporais, mas também de eventos hierárquicos com base nos estados que se encontram ativos na primeira máquina;
- A terceira e quarta máquina não efetuam diretamente a ativação e desativação de pinos de saída, mas permitem efetuar ações de congelamento, descongelamento, desativação ou forçagem de estados das duas primeiras máquinas. Os eventos que provocam as transições dos seus estados são baseados nos valores dos pinos de entrada, tendo associadas ações às suas transições que executam funções de código introduzido diretamente pelo utilizador na *UI*.

#### 4.1.2 Modelação na *interface da aplicação*

A codificação manual do código que descreve o comportamento do problema descrito exigiria a implementação de dezassete estados e dezanove transições, bem como das suas ações e dos eventos que provocam as suas transições. Para além disso, a existência de ações hierárquicas resulta na necessidade de adição de estados de desativação, bem como de mais transições e de *flags*. Sendo assim, é facilmente perceptível que tal implementação seria muito trabalhosa e que uma alteração, por mais simples que seja, no modelo, tal como a adição de um estado intermédio ou a adição de uma ação hierárquica, implicaria a alteração de várias secções desse código.

A aplicação que foi desenvolvida permite modelar e obter o código, que pode ser diretamente implementado num microprocessador/microcontrolador, correspondente à máquina de estados descrita, numa questão de minutos. Para além disso, o projeto pode ser gravado, aberto posteriormente ou alterado, estando sempre à distância de um "click" a obtenção do ficheiro de código com as alterações desejadas. De uma forma geral, a obtenção e implementação do código utilizando a aplicação, pode ser efetuada através das seguintes fases:

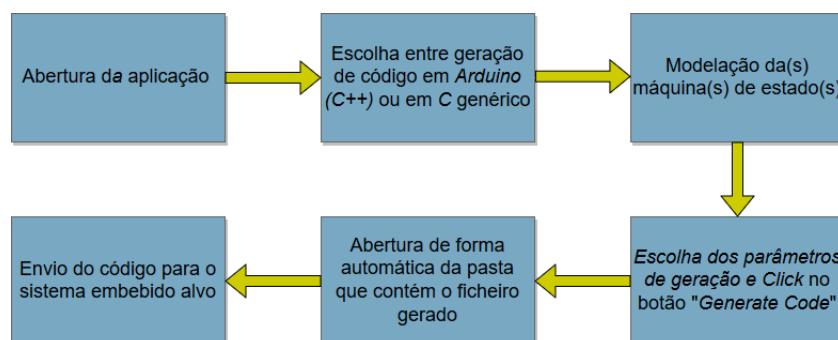


Figura 4.2: Diagrama das fases necessárias para obtenção e implementação do código

De seguida, será dado um exemplo prático das interações que necessitam de ser efetuadas com a *UI*, de forma a obter o código, em *Arduino(C++)*, que corresponda ao comportamento das quatro máquinas que foram descritas neste problema. Relembrando que o código gerado para este problema se encontra no Anexo A.

Uma vez aberta, a aplicação apresenta o seguinte aspeto:

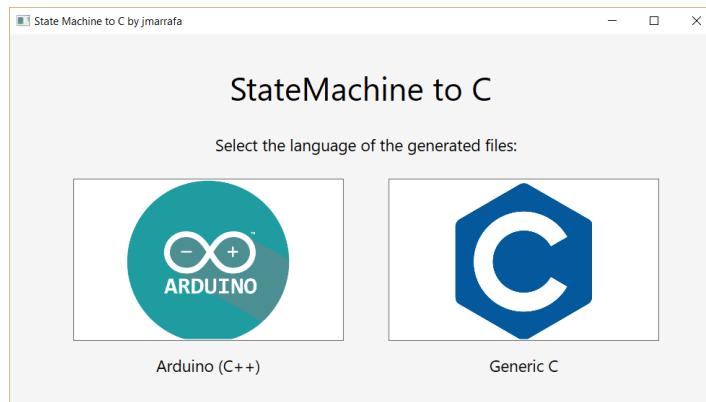


Figura 4.3: *Interface* de escolha da linguagem dos ficheiros gerados

Sendo o microprocessador/microcontrolador alvo deste problema o *ATMega328P* de uma placa *Arduino*, o utilizador deve escolher a opção da esquerda, deparando-se com a seguinte *interface*:

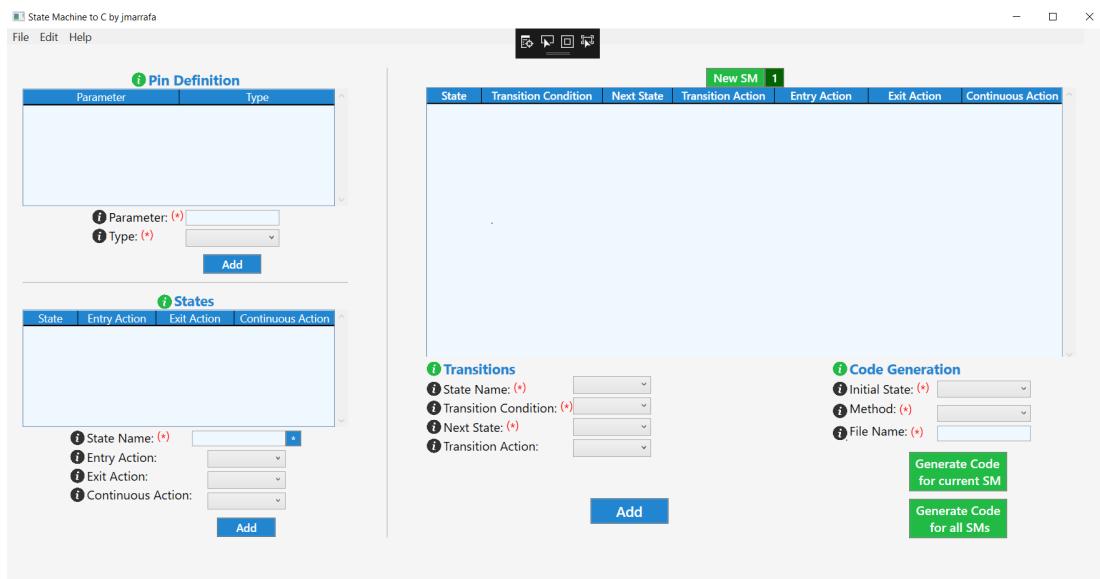


Figura 4.4: *Interface* da aplicação com um projeto vazio

A primeira tabela que deve ser preenchida, de forma a obter o código correspondente ao problema descrito, é a tabela "*Pin Definition*", na qual devem ser introduzidos os parâmetros utilizados pelo *Arduino* como pinos de entrada ou saída, não sendo necessário, nesta fase, especificar os endereços desses pinos. Neste caso, tais pinos consistem em três *LEDs* para a primeira máquina, dois

para a segunda, um botão para a terceira e outro para a quarta. Sendo assim, a tabela apresentaria o seguinte aspecto:

| Pin Definition |           |
|----------------|-----------|
| Parameter      | Type      |
| ledVermelho    | Out Pin   |
| ledAmarelo     | Out Pin   |
| ledVerde       | Out Pin   |
| ledVermelho2   | Out Pin   |
| ledVerde2      | Out Pin   |
| botao1         | Entry Pin |
| botao2         | Entry Pin |

i Parameter: (\*)   
i Type: (\*)

**Add**

Figura 4.5: Tabela "Pin Definition" com os parâmetros do problema 1

Note-se que o preenchimento desta tabela não é um requisito obrigatório. Só devem ser introduzidos parâmetros, caso se pretendam associar a ações ou eventos funcionalidades que se baseiem na ativação, desativação ou verificação dos valores de pinos do *Arduino*. Constatata-se, também que os valores desta tabela são comuns e acessíveis a todas as máquinas de estado presentes no projeto, o que não se verifica nas restantes duas tabelas que são exclusivas à máquina que se encontre selecionada no momento.

O passo seguinte passa pela especificação dos estados que constituem a primeira máquina de estados, bem como as ações que a si se encontram associadas. Sendo assim, a tabela "States" deveria apresentar o seguinte aspecto:

| States  |                      |                       |                   |
|---------|----------------------|-----------------------|-------------------|
| State   | Entry Action         | Exit Action           | Continuous Action |
| stateA1 | ledVermelho set TRUE | ledVermelho set FALSE |                   |
| stateB1 | ledVerde set TRUE    | ledVerde set FALSE    |                   |
| stateC1 | ledAmarelo set TRUE  | ledAmarelo set FALSE  |                   |

i State Name: (\*)  \*  
i Entry Action:    
i Exit Action:    
i Continuous Action:

**Add**

Figura 4.6: Tabela "States" com os parâmetros da primeira máquina

Salienta-se que a ordem de preenchimento das tabelas é importante, no sentido que, caso a primeira tabela não tivesse sido preenchida em primeiro lugar, no momento do preenchimento desta tabela, as opções "*ledVermelho*", "*ledVerde*" e "*ledAmarelo*" não se encontrariam disponíveis.

Neste momento, encontram-se especificados os pinos de entrada e de saída que serão utilizados pelo *Arduino*, tal como os estados e as suas ações que constituem a primeira máquina de estados. No entanto, faltam ainda ser definidas as transições e os eventos que efetuam os seus disparos, assim como quais os estados que por elas serão ligados e a eventual existência de ações associadas a essas mesmas transições.

Para tal é utilizada a tabela "*Transitions*", que no caso da primeira máquina apresentaria o seguinte aspeto:

| New SM 1 |                      |            |                   |                      |                       |             |
|----------|----------------------|------------|-------------------|----------------------|-----------------------|-------------|
| State    | Transition Condition | Next State | Transition Action | Entry Action         | Exit Action           | Continuo... |
| stateA1  | 5000ms               | stateB1    |                   | ledVermelho set TRUE | ledVermelho set FALSE |             |
| stateB1  | 5000ms               | stateC1    |                   | ledVerde set TRUE    | ledVerde set FALSE    |             |
| stateC1  | 1000ms               | stateA1    |                   | ledAmarelo set TRUE  | ledAmarelo set FALSE  |             |

Figura 4.7: Tabela "*Transitions*" com os parâmetros da primeira máquina

Mais uma vez, é de notar a importância da ordem de preenchimento das tabelas, já que as *comboboxes* que permitem associar valores às colunas "*State*" e "*Next State*" só contêm as opções dos estados que se encontram definidos na tabela "*States*".

Uma vez definida a primeira máquina, o seu código poderia ser imediatamente obtido. No entanto, o problema exige a especificação de mais três máquinas. A criação de uma nova máquina dentro do projeto aberto pode ser efetuada com o "click" no botão "*New SM*", que se encontra por cima da tabela de especificação das transições, que pode ser vista na figura 4.7.

Quando uma nova máquina é criada, as tabelas "*States*" e "*Transitions*" encontram-se vazias, devendo, por isso, ser preenchidas com os devidos parâmetros. As tabelas das restantes três máquinas deveriam apresentar o seguinte aspeto:

| States  |                       |                        |          |
|---------|-----------------------|------------------------|----------|
| State   | Entry Action          | Exit Action            | Comments |
| stateA2 | ledVerde2 set TRUE    | ledVerde2 set FALSE    |          |
| stateB2 |                       |                        |          |
| stateC2 | ledVerde2 set TRUE    | ledVerde2 set FALSE    |          |
| stateD2 | ledVermelho2 set TRUE | ledVermelho2 set FALSE |          |

| States  |                         |             |          |
|---------|-------------------------|-------------|----------|
| State   | Entry Action            | Exit Action | Comments |
| stateA3 |                         |             |          |
| stateB3 | SM1-Freeze              |             |          |
| stateC3 | SM1-Unfreeze            |             |          |
| stateD3 | SM1-Deactivate          |             |          |
| stateE3 | SM1-Go to State-stateA1 |             |          |

| States  |                         |             |          |
|---------|-------------------------|-------------|----------|
| State   | Entry Action            | Exit Action | Comments |
| stateA4 |                         |             |          |
| stateB4 | SM2-Freeze              |             |          |
| stateC4 | SM2-Unfreeze            |             |          |
| stateD4 | SM2-Deactivate          |             |          |
| stateE4 | SM2-Go to State-stateA2 |             |          |

Figura 4.8: Tabelas "States" com os parâmetros da segunda, terceira e quarta máquina

| New SM   1   2   3   X |                      |            |                   |                                     |             |
|------------------------|----------------------|------------|-------------------|-------------------------------------|-------------|
| State                  | Transition Condition | Next State | Transition Action | Entry Action                        | Exit Action |
| stateA3                | botao1 rising edge   | stateB3    | counter1++;       |                                     |             |
| stateB3                | botao1 rising edge   | stateC3    | counter1++;       | Hierarchy() SM1-Freeze              |             |
| stateC3                | botao1 rising edge   | stateD3    | counter1++;       | Hierarchy() SM1-Unfreeze            |             |
| stateD3                | botao1 rising edge   | stateE3    | counter1++;       | Hierarchy() SM1-Deactivate          |             |
| stateE3                | botao1 rising edge   | stateA3    | counter1++;       | Hierarchy() SM1-Go to State-stateA1 |             |

| New SM   1   2   3   X |                      |            |                   |                                     |             |
|------------------------|----------------------|------------|-------------------|-------------------------------------|-------------|
| State                  | Transition Condition | Next State | Transition Action | Entry Action                        | Exit Action |
| stateA3                | botao1 rising edge   | stateB3    | counter1++;       |                                     |             |
| stateB3                | botao1 rising edge   | stateC3    | counter1++;       | Hierarchy() SM1-Freeze              |             |
| stateC3                | botao1 rising edge   | stateD3    | counter1++;       | Hierarchy() SM1-Unfreeze            |             |
| stateD3                | botao1 rising edge   | stateE3    | counter1++;       | Hierarchy() SM1-Deactivate          |             |
| stateE3                | botao1 rising edge   | stateA3    | counter1++;       | Hierarchy() SM1-Go to State-stateA1 |             |

| New SM   1   2   3   4   X |                      |            |                   |                                     |             |
|----------------------------|----------------------|------------|-------------------|-------------------------------------|-------------|
| State                      | Transition Condition | Next State | Transition Action | Entry Action                        | Exit Action |
| stateA4                    | botao2 rising edge   | stateB4    | counter2++;       |                                     |             |
| stateB4                    | botao2 rising edge   | stateC4    | counter2++;       | Hierarchy() SM2-Freeze              |             |
| stateC4                    | botao2 rising edge   | stateD4    | counter2++;       | Hierarchy() SM2-Unfreeze            |             |
| stateD4                    | botao2 rising edge   | stateE4    | counter2++;       | Hierarchy() SM2-Deactivate          |             |
| stateE4                    | botao2 rising edge   | stateA4    | counter2++;       | Hierarchy() SM2-Go to State-stateA2 |             |

Figura 4.9: Tabelas "Transitions" com os parâmetros da segunda, terceira e quarta máquina

Após terem sido definidas as quatro máquinas que modelam o comportamento do sistema, o seu código pode ser instantaneamente obtido através da especificação dos seus estados iniciais, seguido do "click" no botão "Generate Code for all SMs", que se distingue do outro botão com a mesma função, por gerar código relativo a todas as máquinas presentes no projeto e não apenas da máquina selecionada no momento do "click".

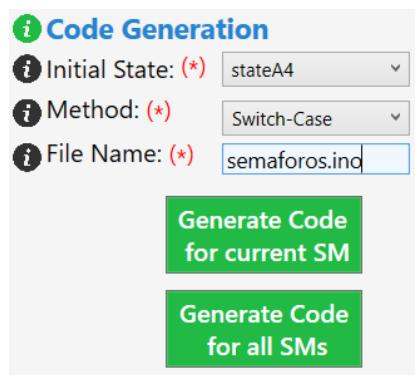


Figura 4.10: Interface da secção para geração do ficheiro de código

É de assinalar que o método de geração da máquina de estados pode assumir a forma "*Switch-Case*", "*If-Else*" ou "*Function Pointers*". A obtenção do código relativo a qualquer um desses formalismos de modelação implica apenas a alteração da opção selecionada na combobox "Method".

O ficheiro gerado, anexado na secção final da dissertação, apresenta cerca de 700 linhas de código e o seu carregamento direto para a placa *Arduino*, será seguido de 2 mensagens de erro.

A primeira mensagem de erro deve-se ao facto de terem sido declarados sete pinos na tabela "*Pin Definition*" que, quando gerados, não têm a si atribuídos endereços. O seguinte excerto de código gerado corresponde à origem da mensagem de erro:

```
//Define all the different inputs addresses
#define btao1 PORTNUMBER //Insert here the address of port
#define btao2 PORTNUMBER //Insert here the address of port
...
//Define all the different outputs addresses
#define ledVermelho PORTNUMBER //Insert here the address of port
#define ledAmarelo PORTNUMBER //Insert here the address of port
#define ledVerde PORTNUMBER //Insert here the address of port
#define ledVermelho2 PORTNUMBER //Insert here the address of port
#define ledVerde2 PORTNUMBER //Insert here the address of port
```

Figura 4.11: Instruções de código geradas por consequência da tabela "*Pin Definition*"

O utilizador deve, por isso, retirar a instrução "*PORTNUMBER*", que se encontra a seguir à definição do parâmetro, e introduzir manualmente o endereço desejado:

```
//Define all the different inputs addresses
#define btao1 7 //Insert here the address of port
#define btao2 8 //Insert here the address of port
...
//Define all the different outputs addresses
#define ledVermelho 11 //Insert here the address of port
#define ledAmarelo 12 //Insert here the address of port
#define ledVerde 13 //Insert here the address of port
#define ledVermelho2 9 //Insert here the address of port
#define ledVerde2 10 //Insert here the address of port
```

Figura 4.12: Introdução dos endereços dos pinos utilizados

O segundo erro deve-se ao facto das ações associadas à transição da terceira e quarta máquina terem sido introduzidas manualmente pelo utilizador na *UI*.

A aplicação trata de gerar todas as variáveis, funcionalidades e instruções correspondentes às especificações do utilizador. No entanto, a funcionalidade "*Code( )*", que pode ser associada a uma ação ou condição de transição, permite dar a liberdade ao utilizador de especificar o código que pretende ver executado na ocorrência dessa ação ou evento, sendo da sua responsabilidade inicializar as variáveis que sejam utilizadas no código por si introduzido. A origem da mensagem de erro, reside no facto de o utilizador ter associado instruções como "*counter1++;*", que é uma variável que não foi inicializada. Adicionando a seguinte instrução de código ao ficheiro gerado, fará com que este compile sem nenhum erro.

```
int counter1 = 0;
int counter2 = 0;
```

Figura 4.13: Inicialização das variáveis utilizadas na função "*Code( )*"

Em suma, todo o código relativo à máquina de estado é gerado de forma automática, sendo apenas necessária a introdução manual dos endereços das portas definidas na tabela "*Pin Definition*" e no caso da utilização da funcionalidade de introdução de código direto "*Code( )*" é da responsabilidade do utilizador efetuar a inicialização das suas variáveis.

#### 4.1.3 Resultados da implementação do código gerado

De forma a testar o código gerado pela aplicação, foi montado um circuito que corresponesse ao problema descrito e introduzido numa placa *Arduino Uno* esse mesmo código.

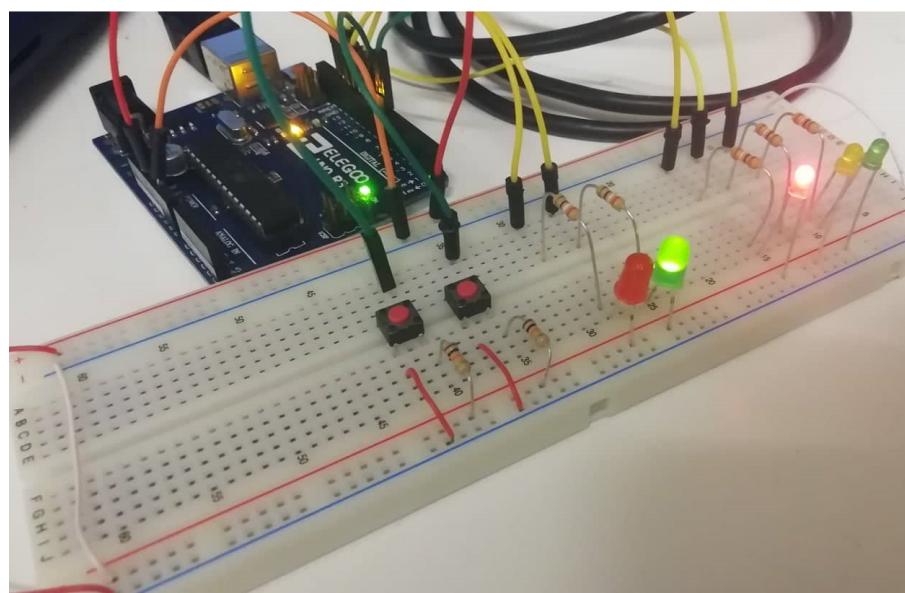


Figura 4.14: Circuito montado para implementação do problema 1

O sistema comportou-se da forma esperada: as quatro máquinas foram inicializadas nos respetivos estados iniciais, as transições temporais dispararam no tempo devido, a alteração de um estado de uma máquina implicou a ocorrência de eventos hierárquicos da máquina que os contém e todos os tipos de ações definidas foram executados de forma correta.

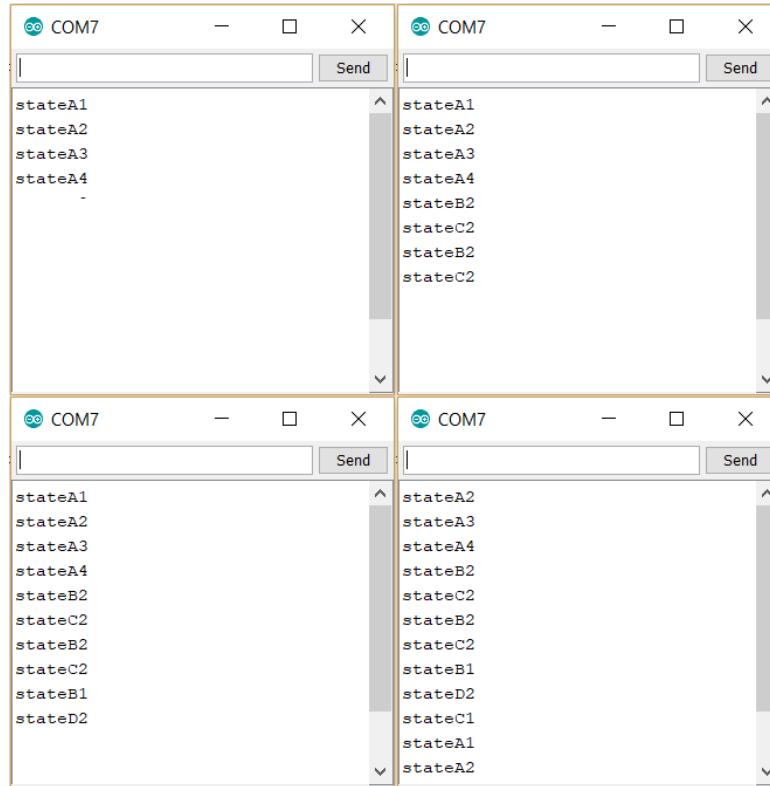


Figura 4.15: Impressão, via porta série, dos estados ativos

O funcionamento normal do sistema pode ser interrompido caso um dos botões seja pressionado, já que o seu *rising edge* implica uma mudança de estado na terceira ou quarta máquina, que à sua entrada tem associadas ações hierárquicas de congelamento, descongelamento, desativação e forçagem de estados da primeira e segunda máquina.

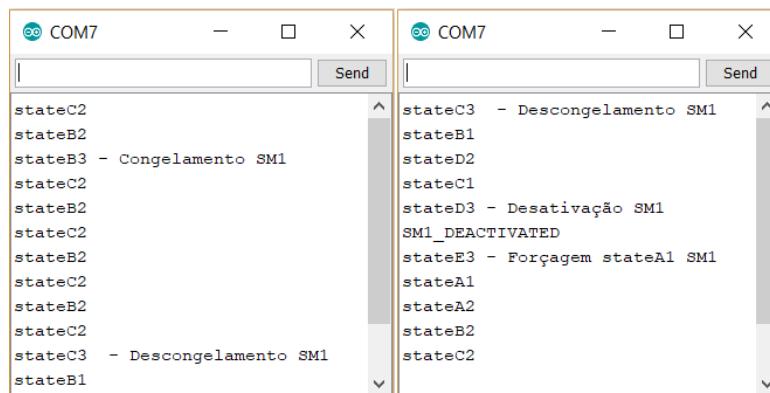


Figura 4.16: Impressão, via porta série, dos estados ativos na ocorrência de ações hierárquicas

## 4.2 Problema 2 - Controlo de uma UPS portátil

### 4.2.1 Descrição do Problema

No segundo problema para demonstração do bom funcionamento da ferramenta desenvolvida, pretendeu-se mostrar que a aplicação pode ser utilizada para problemas de modelação que não envolvam a existência de múltiplas máquinas de estados. No entanto, pretendia-se também implementar e testar o código gerado num problema real e mais complexo. Desta forma, a ferramenta foi utilizada pelo Rafael Gonçalves para gerar o código para a sua dissertação. Numa questão de minutos, foi obtido o ficheiro que correspondia ao comportamento que este precisava de implementar na *UPS* portátil que desenvolveu ao longo do semestre.

A descrição do problema que se segue foi retirada da dissertação [36], podendo esta ser consultada para uma descrição mais detalhada do sistema.

O problema consistiu na modelação de uma máquina de estados que iniciasse num estado "FONTE", representativo do comportamento da *UPS* quando estivesse ligada à fonte de energia. Neste estado o *LED* verde deveria ser aceso, indicando ao utilizador que está a receber energia da rede elétrica. Enquanto isso, os *MOSFETs* 1 e 2 deveriam encontrar-se ativos, permitindo o carregamento da bateria, assim como o seu descarregamento instantâneo quando a *UPS* deixasse de receber energia da fonte.

No caso da *UPS* ser desligada da fonte, o microcontrolador deveria detetar que esta não está a receber energia, transitando para o estado "BATERIA", indicando que está a ser alimentada pela sua bateria através da ativação do *LED* amarelo. Enquanto isso os *MOSFETs* 1 e 2 deveriam permanecer ativos.

Caso o sistema se encontrasse no estado "FONTE" e o microcontrolador detetasse que uma das células excedeu a sua tensão máxima durante o carregamento, ou a temperatura da bateria excedesse os 45°C, o sistema de controlo deveria transitar para o estado "PARA DE CARREGAR". Neste estado o *LED* vermelho deveria estar ativo e o *MOSFET* 1 ser desativado, impedindo o sobrecarregamento da bateria, bem como o seu sobreaquecimento. Para o sistema sair deste estado, a temperatura das células, bem como a sua tensão teriam de baixar, ou a *UPS* ser desligada da fonte.

No caso do sistema de controlo se encontrar no estado "BATERIA", e uma das suas células atingisse os 3V ou a temperatura ultrapassasse os 45°C, o sistema deveria transitar para o estado "PARA DE DESCARREGAR", ligando o *LED* vermelho e desativando os *MOSFETs* 1 e 2, de forma a impedir o descarregamento da bateria ou o seu sobreaquecimento, bem como impedindo que a bateria se carregue quando a *UPS* fosse ligada à rede elétrica, pois a temperatura poderia encontrar-se acima do seu valor máximo. Para sair deste estado, o valor das células teriam de ser superiores a 3V e a temperatura necessitava de baixar para um valor inferior a 45°C. Caso a *UPS* fosse ligada à fonte, deveria ser dado inicio ao seu carregamento através da ativação do *MOSFET* 1 no estado "FONTE".

Na figura 4.17 pode ser vista a máquina de estados que foi modelada, segundo a norma *UML*, e introduzida na aplicação de forma a obter o código que corresponesse ao problema descrito.

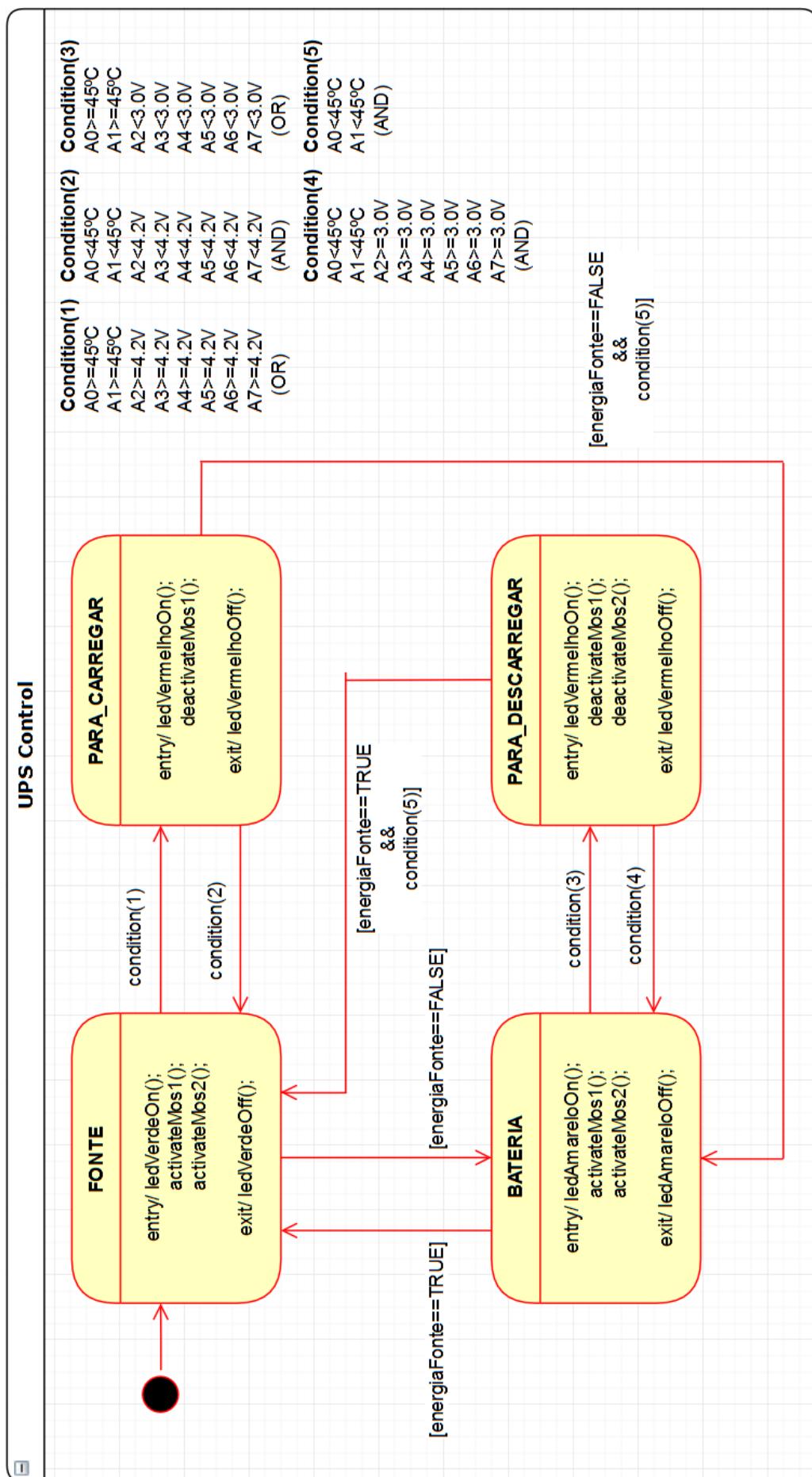


Figura 4.17: UML statechart para controlo de uma UPS portátil

### 4.2.2 Modelação na interface da aplicação

Mais uma vez relembra-se o diagrama das fases necessárias para a obtenção e implementação do código na aplicação:

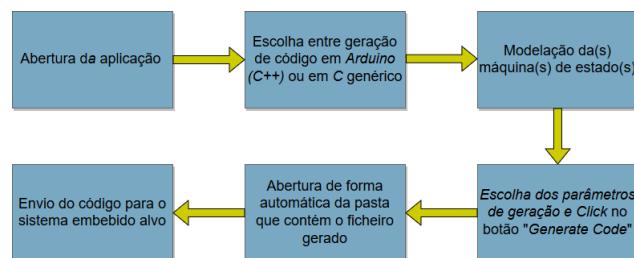


Figura 4.18: Diagrama das fases necessárias para obtenção e implementação do código

De forma a obter o código correspondente ao problema descrito, uma vez aberta a aplicação foi escolhida a opção *Arduino (C++)*.

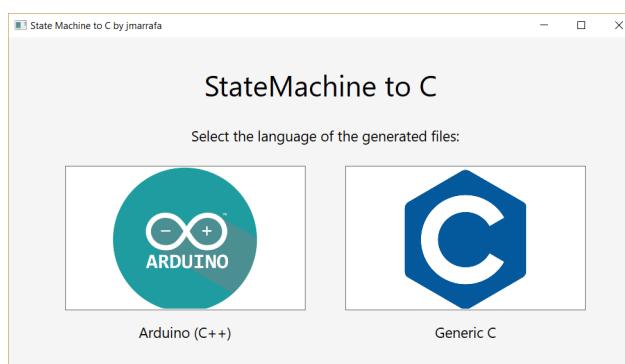


Figura 4.19: Interface de escolha da linguagem dos ficheiros gerados

Na tabela "*Pin Definition*" foram introduzidos os parâmetros utilizados pelo *Arduino* como pinos de entrada/saída, não sendo necessário, nesta fase, especificar os endereços desses pinos. Neste caso, analisando a modelação efetuada na figura 4.17, os parâmetros de saída consistem em três *LEDs*, o *MOS1* e *2*. Existe apenas um parâmetro de entrada com o nome "energiaFonte".

| Pin Definition |           |
|----------------|-----------|
| Parameter      | Type      |
| ledAmarelo     | Out Pin   |
| ledVerde       | Out Pin   |
| ledVermelho    | Out Pin   |
| MOS1           | Out Pin   |
| MOS2           | Out Pin   |
| energiaFonte   | Entry Pin |

Parâmetros adicionados:

- Parameter: (\*) energiaFonte
- Type: (\*) Entry Pin

Botões: Add

Figura 4.20: Tabela *Pin definition* com os parâmetros do problema 2

Posteriormente, foram introduzidos na tabela "States", os estados que constituem a máquina, assim como as ações a si associadas. Neste caso, todos os estados contêm ações à sua entrada e saída.

A aplicação contém a funcionalidade de ativação/desativação de pinos, que pode ser associada a uma ação. Assim sendo, considerando o estado "FONTE", a sua ação de saída é a desativação do ledVerde, tal foi definido da seguinte forma:

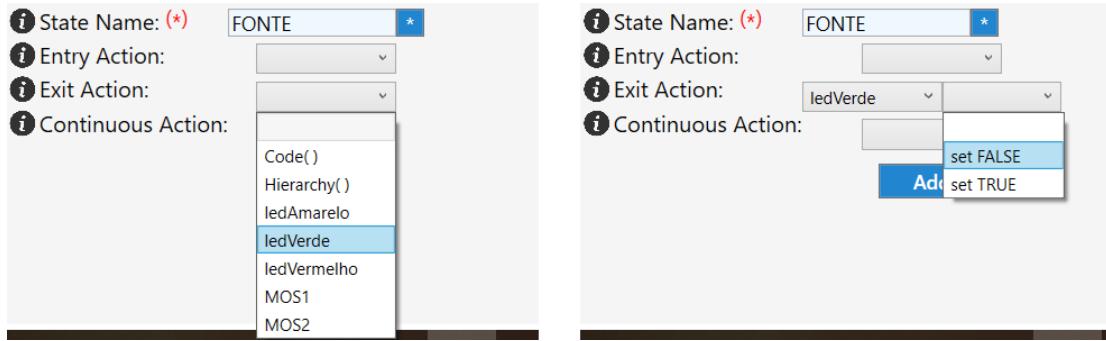


Figura 4.21: Associação de funções com um único parâmetro a uma ação de saída

No caso das ações de entrada do estado em questão, estas consistem na ativação de três parâmetros. Se existisse apenas um parâmetro, como na situação da ação de saída, tal poderia ser efetuado da forma indicada. No entanto, a existência de múltiplos parâmetros utilizados na mesma ação, implicou a necessidade de uma introdução manual das instruções, usando a função *Code*:

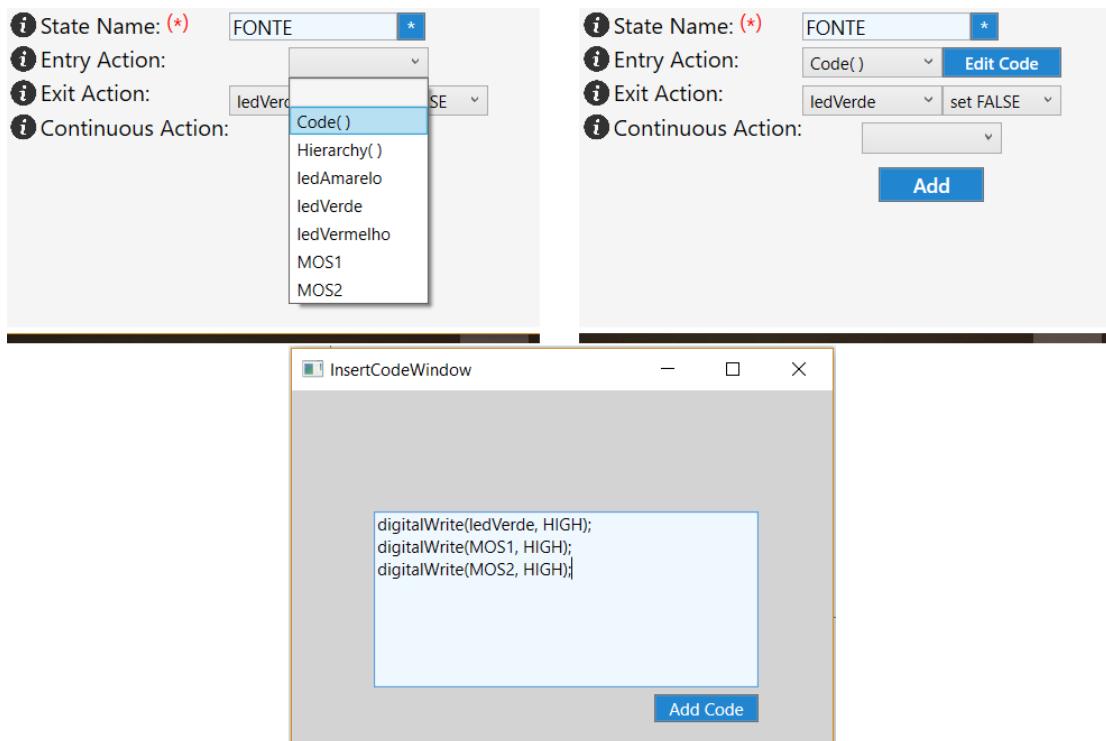


Figura 4.22: Associação de funções com múltiplos parâmetros a uma ação de entrada

Nas seguintes figuras pode-se verificar que foi efetuado o mesmo procedimento para os res- tantes estados:



Figura 4.23: Associação de funções com múltiplos parâmetros a múltiplas ações de entrada

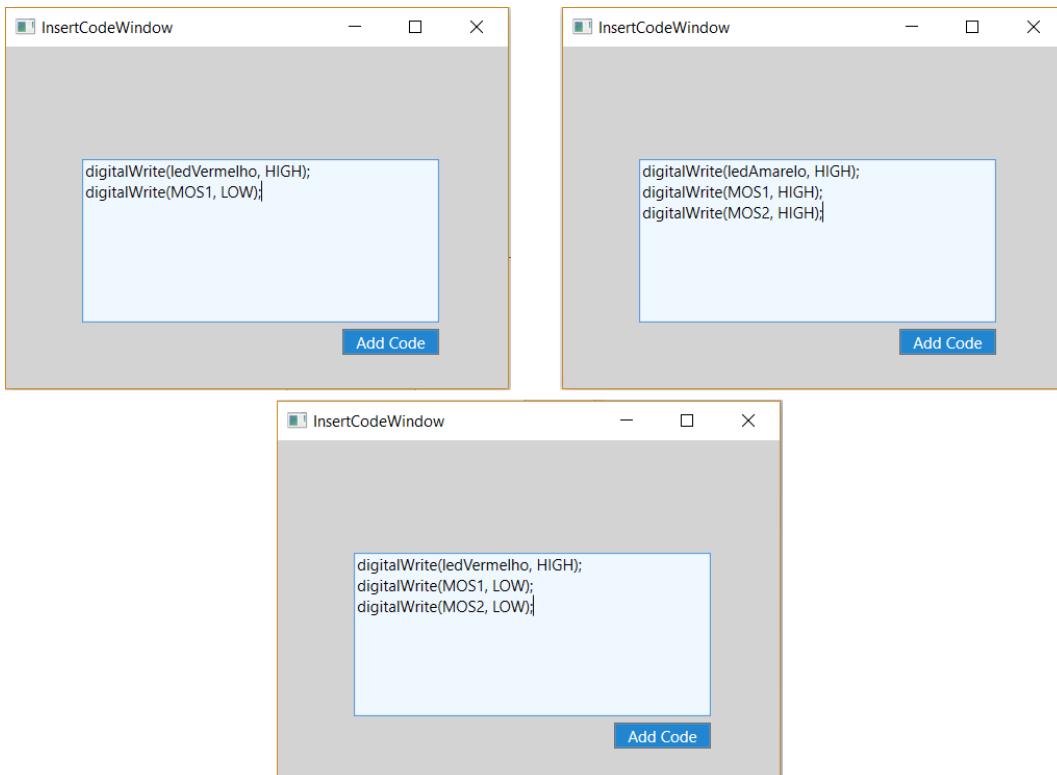


Figura 4.24: Associação de funções com múltiplos parâmetros a múltiplas ações de entrada

Estando os estados definidos, estes puderam ser selecionados para definição das transições. Tal como nas ações, quando o evento que dispara a transição consiste apenas num parâmetro, como é o caso de duas transições da máquina a ser modelada, tal pode ser especificado da seguinte forma:

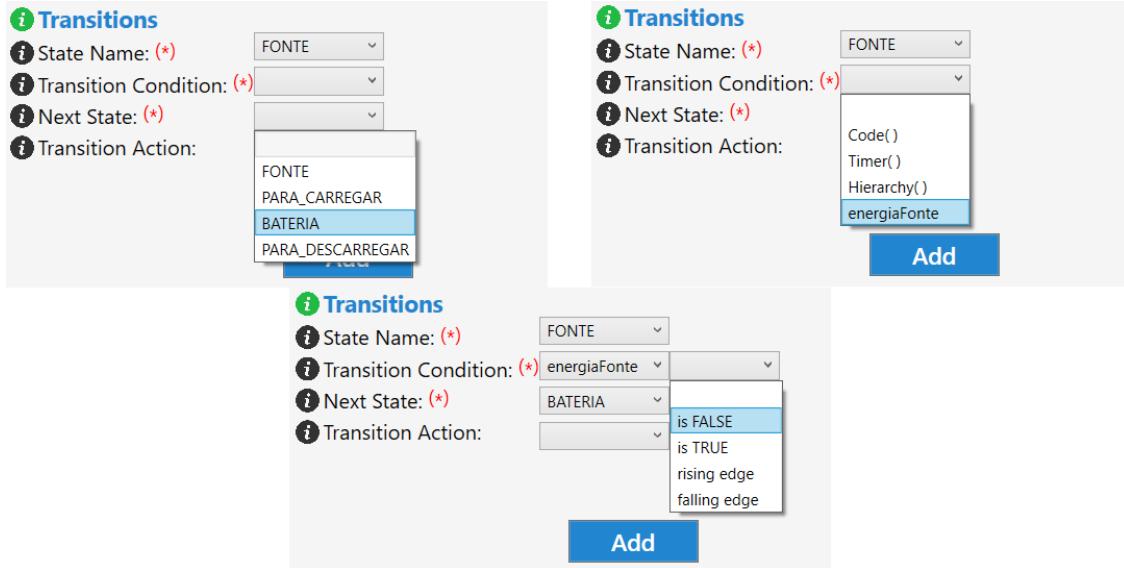


Figura 4.25: Associação de eventos com um único parâmetro a uma transição

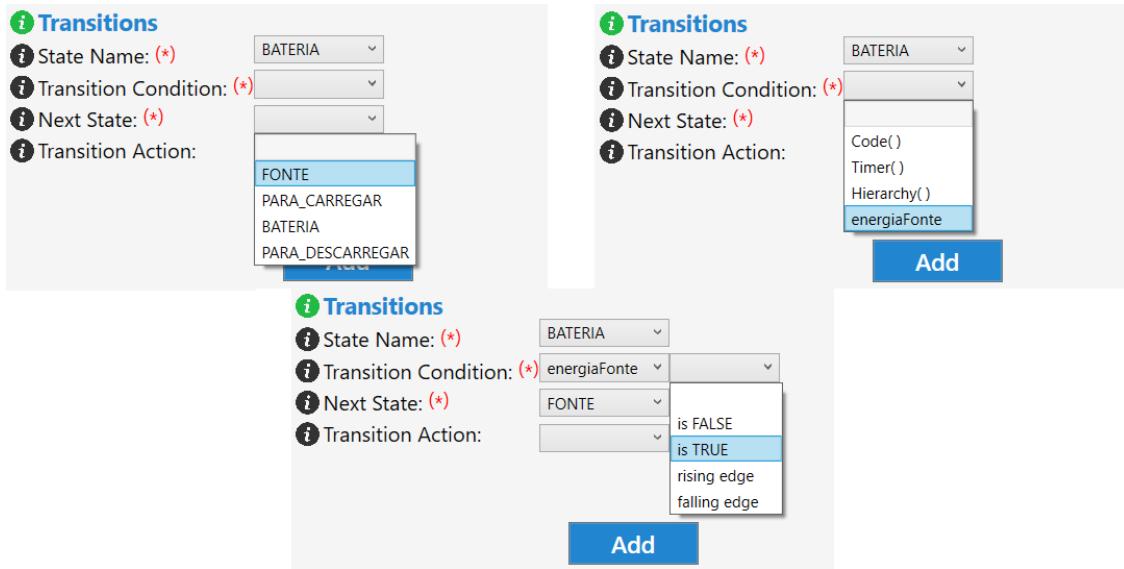


Figura 4.26: Associação de eventos com um único parâmetro a uma transição

Quando a condição de transição envolve mais do que um parâmetro ou depende de uma instrução de código específica, então deve ser utilizada a opção *Code*:

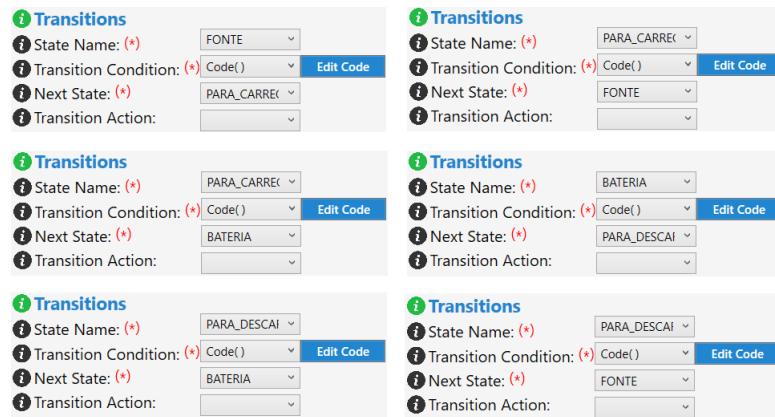


Figura 4.27: Associação de eventos com múltiplos parâmetros a múltiplas transições



Figura 4.28: Associação de eventos com múltiplos parâmetros a múltiplas transições

Cumpridos estes passos, a máquina de estados encontrava-se modelada e o seu código pôde ser obtido através do preenchimento da secção de geração:

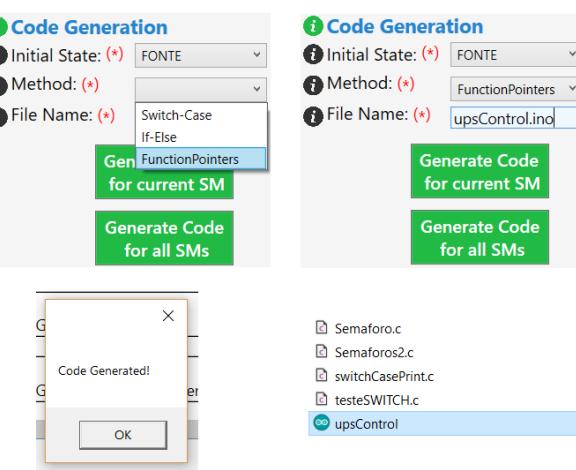


Figura 4.29: Geração do código para o problema 2

Uma vez aberto o ficheiro de código gerado, foi necessário preencher as instruções de definição dos endereços dos pinos com os endereços desejados.

```
//Define all the different inputs addresses
#define energiaFonte 10 //Insert here the address of port

//Define all the different outputs addresses

#define ledAmarelo 6 //Insert here the address of port
#define ledVerde 7 //Insert here the address of port
#define ledVermelho 13 //Insert here the address of port
#define MOS1 9 //Insert here the address of port
#define MOS2 8 //Insert here the address of port
```

Figura 4.30: Introdução manual dos endereços dos pinos do *Arduino* utilizados

Tendo sido necessário também, efetuar a inicialização as variáveis ou funções utilizadas na funcionalidade de introdução de código manualmente *Code*. Neste caso foi utilizada nas condições de transição uma função *condition()*. Portanto, teve de ser acrescentado ao ficheiro gerado uma definição dessa função, que retornava o valor *true*, caso se verificassem as condições que se encontram presentes na legenda da máquina de estados modelada para este problema, figura 4.17.

Relembra-se que qualquer projeto criado na aplicação pode ser gravado e posteriormente aberto e editado, estando sempre à distância de um "click" a obtenção instantânea do código com as alterações que sejam necessárias efetuar ao sistema de controlo.

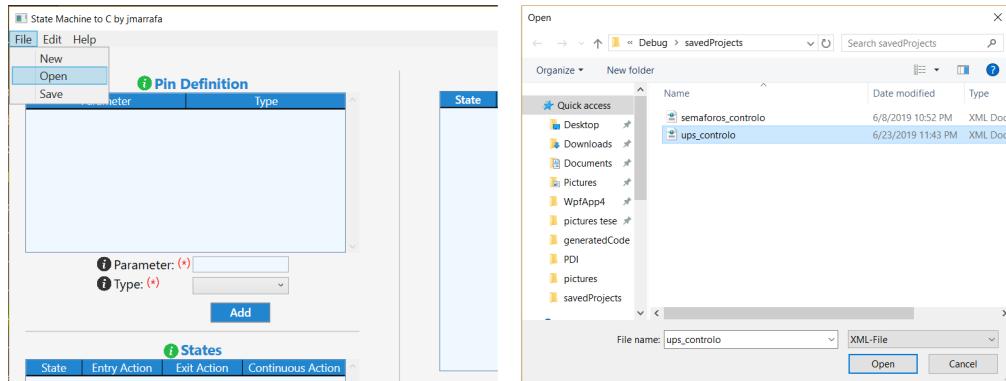


Figura 4.31: Abertura do projeto criado

#### 4.2.3 Resultados da implementação do código gerado

Utilizando o código gerado pela aplicação desenvolvida no âmbito desta dissertação, foi possível obter o comportamento desejado para o funcionamento do sistema, a partir daí, o autor da *UPS* em questão, desenvolveu todas funcionalidades extra que pretendia para as ações dos estados, sempre com a base comportamental da máquina de estados gerada.



Figura 4.32: Montagem da *UPS* portátil

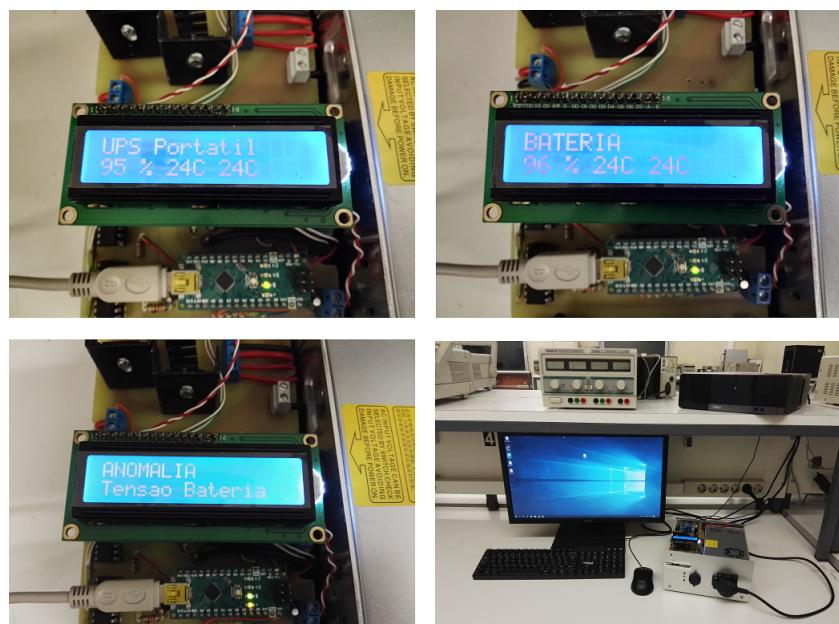


Figura 4.33: Resultados obtidos



# Capítulo 5

## Conclusões e Trabalho Futuro

### 5.1 Cumprimento dos objetivos

O principal objetivo do projeto visava o desenvolvimento de uma aplicação que permitisse modelar máquinas de estados e obter o seu código para ser implementado diretamente em sistemas embebidos e este foi realizado com sucesso.

Os testes que foram efetuados provaram que a aplicação é funcional e intuitiva, mostrando que a sua utilização permite, efetivamente, obter o código que corresponda ao modelo especificado. Para além disso, a obtenção desse código é altamente customizável, já que o utilizador tem ao seu dispor:

- A escolha entre as linguagens *Arduino* (*C++*) e *C*;
- Quais dos diferentes métodos de estruturação de código pretende (*switch-case*, *if-else* e *function-pointers*) utilizar;
- Se pretende gerar código apenas de uma máquina ou de todas as máquinas presentes no projeto;
- A escolha do nome dos estados e dos parâmetros que são utilizados, por exemplo, para funções de ativação/desativação de pinos;
- A possibilidade de introdução manual de instruções de código na *interface* da aplicação que podem ser associadas às condições das transições ou às ações executadas, bem como o nome dos ficheiros gerados;

Aliado a esse alto nível de customização vem a possibilidade de especificação de grande parte dos formalismos vistos no capítulo 2, desde ações à entrada, saída ou continuamente num estado, ações executadas numa transição e a possibilidade de existência de mais do que uma máquina de estados no projeto, utilizando o conceito das regiões ortogonais.

Suporta também funcionalidades de impedimento de tentativas de introdução de comportamentos que não cumpram a norma de construção de uma máquina de estados e é possível a sua reutilização, pois os projetos nela criados podem ser editados, gravados e abertos posteriormente.

Para além disso, a aplicação não se delimita às funcionalidades das normas de modelação existentes, fugindo do comum e introduzindo a possibilidade de execução de ações que são utilizadas em modelos de *GRAFCET*, as macro-ações que no caso da aplicação desenvolvida, permitem congelar, descongelar, desativar e forçar um estado de outra máquina existente no projeto.

O facto de se ter optado por uma modelação tabelar, em detrimento de uma modelação gráfica, permitiu focar nas funcionalidades da camada *back-end* e de geração de código, o que se revelou ser uma mais valia, já que para além de tornar pouco confusa a fase de modelação da máquina de estados, permitiu desenvolver todas estas funcionalidades.

As características mencionadas tornam a ferramenta desenvolvida única, já que não existem muitos *softwares* de modelação de máquinas de estados e de obtenção do seu código orientado à programação embebida e dentro dos que existem, nenhum apresenta todas as funcionalidades desta aplicação. Não querendo com isto dizer que a aplicação desenvolvida é melhor ou pior, aliás se formos a analisar o número total de funcionalidades que os *softwares* mencionados na secção 2.5 dispõe, verificamos um número muito maior de funcionalidades. No entanto, estes *softwares* foram desenvolvidos em ambiente empresarial, por equipas de engenheiros e já existem no mercado há anos.

## 5.2 Trabalho Futuro

Quanto ao trabalho futuro que pode ser desenvolvido para melhorar a aplicação, este passa pela adição de mais funcionalidades da norma *UML*, nomeadamente a possibilidade de associação de estados compostos, que contêm outros estados e, nesse caso, deve ser adicionada a possibilidade de uma modelação de forma gráfica, na qual seja possível modelar a máquina através de estados e transições desenhadas diretamente na *interface*, bem como a possibilidade de expandir os estados compostos, associar os tipos de ações e eventos que já foram desenvolvidos nesta dissertação e, aproveitando os mecanismos de geração de código que já se encontram criados, mantendo sempre a possibilidade do utilizador escolher a modelação tabelar já desenvolvida.

Incorporada na opção da modelação gráfica pode ser elaborada uma ferramenta de *debug*, na qual seja possível comunicar com o microprocessador/microcontrolador e obter indicações no grafismo de quais os estados que se encontram ativos no momento, qual a transição que lá o levou e que ações estão a ser executadas.

Também pode ser acrescentada a possibilidade de distinção entre uma *self-transition* e uma *internal transition*, já que neste momento a aplicação só suporta a primeira opção, bem como a reformulação dos ficheiros *XML* que contêm os projetos gravados da aplicação para *SCXML*.

Podem ainda ser adicionadas mais opções de linguagens para o código gerado, tais como o *Verilog*, *VHDL*, *Java* ou qualquer linguagem mencionada na secção 2.3. Podendo também ser reforçada a possibilidade de especificação da marca e do modelo do sistema embebido alvo, disponibilizando funções pré-definidas associadas as ações e eventos que se destinem a imprimir instruções características desse mesmo sistema, tal como foi efetuado neste projeto para o caso do *Arduino*.



## Anexo A

# Anexos

### A.1 Código gerado pela aplicação para o Problema 1

```
1 /*  
2  Code generated using StateMachine2C by jmarrafa  
3  For support contact josemarrafa3@gmail.com  
4  FEUP 2019  
5 */  
6  
7 int INIT1=0;  
8 int INIT2=0;  
9 int INIT3=0;  
10 int INIT4=0;  
11  
12 // Declare possible states  
13 typedef enum  
14 {  
15     stateA1 ,  
16     stateB1 ,  
17     stateC1 ,  
18     SM1_DEACTIVATED,  
19     stateA2 ,  
20     stateB2 ,  
21     stateC2 ,  
22     stateD2 ,  
23     SM2_DEACTIVATED,  
24     stateA3 ,  
25     stateB3 ,  
26     stateC3 ,  
27     stateD3 ,  
28     stateE3 ,  
29     stateA4 ,  
30     stateB4 ,  
31     stateC4 ,  
32     stateD4 ,
```

```
33     stateE4 ,  
34     NUM_STATES  
35 } stateNames ;  
36  
37 // Define all the different inputs addresses  
38 #define bota01 7  
39 #define bota02 8  
40 // Define all the different outputs addresses  
41 #define ledVermelho 11  
42 #define ledAmarelo 12  
43 #define ledVerde 13  
44 #define ledVermelho2 9  
45 #define ledVerde2 10  
46  
47 //Aux variables for transitions that need rising edge detection  
48 int lastStatebota01=0;  
49 int lastStatebota02=0;  
50  
51 // Variables initialized manually  
52 int counter1 = 0;  
53 int counter2 = 0;  
54  
55 void setup()  
{  
56     pinMode(bota01 , INPUT);  
57     pinMode(bota02 , INPUT);  
58  
59     pinMode(ledVermelho , OUTPUT);  
60     pinMode(ledAmarelo , OUTPUT);  
61     pinMode(ledVerde , OUTPUT);  
62     pinMode(ledVermelho2 , OUTPUT);  
63     pinMode(ledVerde2 , OUTPUT);  
64  
65     Serial.begin(9600);  
66 }  
67  
68  
69 // Declare the initial states of the SMs  
70 stateNames currentState1 = stateA1 ;  
71 stateNames currentState2 = stateA2 ;  
72 stateNames currentState3 = stateA3 ;  
73 stateNames currentState4 = stateA4 ;  
74  
75 stateNames forcedcurrentState1 ;  
76 stateNames forcedcurrentState2 ;  
77  
78 // Declare the timers to be used  
79 unsigned long timerState1 = 0;  
80 unsigned long previousMillis1 = 0;  
81
```

```
82 unsigned long timerState2 = 0;
83 unsigned long previousMillis2 = 0;
84
85 unsigned long timerState3 = 0;
86 unsigned long previousMillis3 = 0;
87
88 unsigned long timerState4 = 0;
89 unsigned long previousMillis4 = 0;
90
91 const long timerA1 = 5000;
92 const long timerB1 = 5000;
93 const long timerC1 = 1000;
94 const long timerA2 = 3000;
95 const long timerB2 = 500;
96 const long timerC2 = 500;
97
98 bool deactivateSM1 = false;
99 bool freezeSM1 = false;
100 bool forceStateSM1 = false;
101
102 bool deactivateSM2 = false;
103 bool freezeSM2 = false;
104 bool forceStateSM2 = false;
105
106 void loop()
107 {
108     runSMs();
109 }
110
111 void runSMs()
112 {
113     unsigned long currentMillis = millis();
114
115     // State Machine 1
116     switch(currentState1)
117     {
118         case stateA1:
119
120             if (INIT1 == 0)
121             {
122                 Serial.println("stateA1");
123                 previousMillis1 = currentMillis;
124                 digitalWrite(ledVermelho, HIGH); //on entry action
125                 INIT1=1;
126             }
127
128             timerState1 = currentMillis - previousMillis1;
129
130             if ((timerState1 >= timerA1) && (!freezeSM1)) // transition condition
```

```
131 {
132     currentState1 = stateB1;
133     digitalWrite(ledVermelho, LOW); //exit action
134     INIT1 = 0; //resets the init indicator
135 }
136
137 //Transition conditions as a result of hierarchichal actions
138 if(deactivateSM1 == true)
139 {
140     currentState1 = SMI_DEACTIVATED;
141     digitalWrite(ledVermelho, LOW); //exit action
142     INIT1 = 0;
143 }
144
145 if(forceStateSM1 == true)
146 {
147     currentState1 = forcedCurrentState1;
148     forceStateSM1 = false;
149     digitalWrite(ledVermelho, LOW); //exit action
150     INIT1 = 0;
151 }
152
153 break;
154
155 case stateB1:
156
157 if(INIT1 == 0)
158 {
159     Serial.println("stateB1");
160     previousMillis1 = currentMillis;
161     digitalWrite(ledVerde, HIGH); //on entry action
162     INIT1=1;
163 }
164
165 timerState1 = currentMillis - previousMillis1;
166
167 if ((timerState1 >= timerB1) && (!freezeSM1)) //transition condition
168 {
169     currentState1 = stateC1;
170     digitalWrite(ledVerde, LOW); //exit action
171     INIT1 = 0; //resets the init indicator
172 }
173
174 //Transition conditions as a result of hierarchichal actions
175 if(deactivateSM1 == true)
176 {
177     currentState1 = SMI_DEACTIVATED;
178     digitalWrite(ledVerde, LOW); //exit action
179     INIT1 = 0;
```

```
180     }
181
182     if(forceStateSM1 == true)
183     {
184         currentState1 = forcedCurrentState1;
185         forceStateSM1 = false;
186         digitalWrite(ledVerde, LOW); // exit action
187         INIT1 = 0;
188     }
189
190     break;
191
192 case stateC1:
193
194     if(INIT1 == 0)
195     {
196         Serial.println("stateC1");
197         previousMillis1 = currentMillis;
198         digitalWrite(ledAmarelo, HIGH); //on entry action
199         INIT1=1;
200     }
201
202     timerState1 = currentMillis - previousMillis1;
203
204     if ((timerState1 >= timerC1) && (!freezeSM1)) // transition condition
205     {
206         currentState1 = stateA1;
207         digitalWrite(ledAmarelo, LOW); //exit action
208         INIT1 = 0; //resets the init indicator
209     }
210
211 // Transition conditions as a result of hierachical actions
212 if(deactivateSM1 == true)
213 {
214     currentState1 = SM1_DEACTIVATED;
215     digitalWrite(ledAmarelo, LOW); //exit action
216     INIT1 = 0;
217 }
218
219 if(forceStateSM1 == true)
220 {
221     currentState1 = forcedCurrentState1;
222     forceStateSM1 = false;
223     digitalWrite(ledAmarelo, LOW); //exit action
224     INIT1 = 0;
225 }
226
227 break;
228 }
```

```
229     case SM1_DEACTIVATED:
230
231         if(forceStateSM1 == true)
232     {
233             currentState1 = forcedCurrentState1;
234             forceStateSM1 = false;
235             deactivateSM1 = false;
236         }
237         break;
238
239     default: break;
240 }
241
242 // State Machine 2
243 switch(currentState2)
244 {
245     case stateA2:
246
247         if(INIT2 == 0)
248     {
249             Serial.println("stateA2");
250             previousMillis2 = currentMillis;
251             digitalWrite(ledVerde2, HIGH); //on entry action
252             INIT2=1;
253         }
254
255         timerState2 = currentMillis - previousMillis2;
256
257         if ((timerState2 >= timerA2) && (!freezeSM2)) // transition condition
258     {
259             currentState2 = stateB2;
260             digitalWrite(ledVerde2, LOW); // exit action
261             INIT2 = 0; // resets the init indicator
262         }
263
264
265 // Transition conditions as a result of hierachichal actions
266 if(deactivateSM2 == true)
267 {
268     currentState2 = SM2_DEACTIVATED;
269     digitalWrite(ledVerde2, LOW); // exit action
270     INIT2 = 0;
271 }
272
273 if(forceStateSM2 == true)
274 {
275     currentState2 = forcedCurrentState2;
276     forceStateSM2 = false;
277     digitalWrite(ledVerde2, LOW); // exit action
```

```

278     INIT2 = 0;
279 }
280
281     break;
282
283 case stateB2:
284
285     if (INIT2 == 0)
286     {
287         Serial.println("stateB2");
288         previousMillis2 = currentMillis;
289         INIT2=1;
290     }
291
292     timerState2 = currentMillis - previousMillis2;
293
294     if ((timerState2 >= timerB2) && (!freezeSM2)) // transition condition
295     {
296         currentState2 = stateC2;
297         INIT2 = 0; //resets the init indicator
298     }
299
300     if ((currentState1 == stateB1)&& (!freezeSM2))
301     {
302         currentState2 = stateD2;
303         INIT2 = 0; //resets the init indicator
304     }
305
306 // Transition conditions as a result of hierachical actions
307     if(deactivateSM2 == true)
308     {
309         currentState2 = SM2_DEACTIVATED;
310         INIT2 = 0;
311     }
312
313     if (forceStateSM2 == true)
314     {
315         currentState2 = forcedCurrentState2;
316         forceStateSM2 = false;
317         INIT2 = 0;
318     }
319
320     break;
321
322 case stateC2:
323
324     if (INIT2 == 0)
325     {
326         Serial.println("stateC2");

```

```
327     previousMillis2 = currentMillis;
328     digitalWrite(ledVerde2, HIGH); //on entry action
329     INIT2=1;
330 }
331
332     timerState2 = currentMillis - previousMillis2;
333
334     if ((timerState2 >= timerC2) && (!freezeSM2)) // transition condition
335     {
336         currentState2 = stateB2;
337         digitalWrite(ledVerde2, LOW); //exit action
338         INIT2 = 0; //resets the init indicator
339     }
340
341     if ((currentState1 == stateB1)&& (!freezeSM2))
342     {
343         currentState2 = stateD2;
344         digitalWrite(ledVerde2, LOW); //exit action
345         INIT2 = 0; //resets the init indicator
346     }
347
348 // Transition conditions as a result of hierarchichal actions
349     if(deactivateSM2 == true)
350     {
351         currentState2 = SM2_DEACTIVATED;
352         digitalWrite(ledVerde2, LOW); //exit action
353         digitalWrite(ledVerde2, LOW); //exit action
354         INIT2 = 0;
355     }
356
357     if(forceStateSM2 == true)
358     {
359         currentState2 = forcedCurrentState2;
360         forceStateSM2 = false;
361         digitalWrite(ledVerde2, LOW); //exit action
362         digitalWrite(ledVerde2, LOW); //exit action
363         INIT2 = 0;
364     }
365
366     break;
367
368 case stateD2:
369
370     if(INIT2 == 0)
371     {
372         Serial.println("stateD2");
373         previousMillis2 = currentMillis;
374         digitalWrite(ledVermelho2, HIGH); //on entry action
375         INIT2=1;
```

```

376     }
377
378     timerState2 = currentMillis - previousMillis2;
379
380     if ((currentState1 == stateA1)&& (!freezeSM2))
381     {
382         currentState2 = stateA2;
383         digitalWrite(ledVermelho2, LOW); // exit action
384         INIT2 = 0; // resets the init indicator
385     }
386
387     // Transition conditions as a result of hierachical actions
388     if(deactivateSM2 == true)
389     {
390         currentState2 = SM2_DEACTIVATED;
391         digitalWrite(ledVermelho2, LOW); // exit action
392         INIT2 = 0;
393     }
394
395     if(forceStateSM2 == true)
396     {
397         currentState2 = forcedCurrentState2;
398         forceStateSM2 = false;
399         digitalWrite(ledVermelho2, LOW); // exit action
400         INIT2 = 0;
401     }
402
403     break;
404
405 case SM2_DEACTIVATED:
406
407     if(forceStateSM2 == true)
408     {
409         currentState2 = forcedCurrentState2;
410         forceStateSM2 = false;
411         deactivateSM2 = false;
412     }
413
414     break;
415
416 default: break;
417 }
418
419 // State Machine 3
420 switch(currentState3)
421 {
422     case stateA3:
423
424         if(INIT3 == 0)

```

```
425 {
426     Serial.println("stateA3");
427     previousMillis3 = currentMillis;
428     INIT3=1;
429 }
430
431 timerState3 = currentMillis - previousMillis3;
432
433 if((digitalRead(botao1) != lastStatebotao1)) // transition condition (
434 rising edge)
435 {
436     if((digitalRead(botao1) == HIGH)) // transition condition (rising edge)
437     {
438         counter1++; //on transition action
439         currentState3 = stateB3;
440         INIT3 = 0; // resets the init indicator
441     }
442     lastStatebotao1 = digitalRead(botao1);
443     delay(20); // to avoid bouncing (optional)
444 }
445
446 break;
447
448 case stateB3:
449
450 if(INIT3 == 0)
451 {
452     Serial.println("stateB3");
453     previousMillis3 = currentMillis;
454     freezeSM1 = true; //on entry action
455     INIT3=1;
456 }
457
458 timerState3 = currentMillis - previousMillis3;
459
460 if((digitalRead(botao1) != lastStatebotao1)) // transition condition (
461 rising edge)
462 {
463     if((digitalRead(botao1) == HIGH)) // transition condition (rising edge)
464     {
465         counter1++; //on transition action
466         currentState3 = stateC3;
467         INIT3 = 0; // resets the init indicator
468     }
469     lastStatebotao1 = digitalRead(botao1);
470     delay(20); // to avoid bouncing (optional)
471 }
472
473 break;
```

```
472
473     case stateC3 :
474
475         if (INIT3 == 0)
476     {
477             Serial.println("stateC3");
478             previousMillis3 = currentMillis;
479             freezeSM1 = false; //on entry action
480             INIT3=1;
481         }
482
483         timerState3 = currentMillis - previousMillis3;
484
485         if ((digitalRead(botao1) != lastStatebotao1)) // transition condition (rising edge)
486         {
487             if ((digitalRead(botao1) == HIGH)) // transition condition (rising edge)
488             {
489                 counter1++; //on transition action
490                 currentState3 = stateD3;
491                 INIT3 = 0; //resets the init indicator
492             }
493             lastStatebotao1 = digitalRead(botao1);
494             delay(20); //to avoid bouncing (optional)
495         }
496
497         break;
498
499     case stateD3 :
500
501         if (INIT3 == 0)
502     {
503             Serial.println("stateD3");
504             previousMillis3 = currentMillis;
505             deactivateSM1 = true; //on entry action
506             INIT3=1;
507         }
508
509         timerState3 = currentMillis - previousMillis3;
510
511         if ((digitalRead(botao1) != lastStatebotao1)) // transition condition (rising edge)
512         {
513             if ((digitalRead(botao1) == HIGH)) // transition condition (rising edge)
514             {
515                 counter1++; //on transition action
516                 currentState3 = stateE3;
517                 INIT3 = 0; //resets the init indicator
518             }
```

```
519     lastStatebotao1 = digitalRead(botao1);
520     delay(20); // to avoid bouncing (optional)
521 }
522
523     break;
524
525 case stateE3:
526
527     if(INIT3 == 0)
528     {
529         Serial.println("stateE3");
530         previousMillis3 = currentMillis;
531         forcedCurrentState1 = stateA1;
532         forceStateSM1 = true;
533         INIT3=1;
534     }
535
536     timerState3 = currentMillis - previousMillis3;
537
538     if((digitalRead(botao1) != lastStatebotao1)) // transition condition (rising edge)
539     {
540         if((digitalRead(botao1) == HIGH)) // transition condition (rising edge)
541         {
542             counter1++; //on transition action
543             currentState3 = stateA3;
544             INIT3 = 0; // resets the init indicator
545         }
546         lastStatebotao1 = digitalRead(botao1);
547         delay(20); // to avoid bouncing (optional)
548     }
549
550     break;
551
552 default: break;
553 }
554
555 // State Machine 4
556 switch(currentState4)
557 {
558 case stateA4:
559
560     if(INIT4 == 0)
561     {
562         Serial.println("stateA4");
563         previousMillis4 = currentMillis;
564         INIT4=1;
565     }
566 }
```

```
567     timerState4 = currentMillis - previousMillis4;  
568  
569     if((digitalRead(botao2) != lastStatebotao2)) // transition condition (rising edge)  
570     {  
571         if((digitalRead(botao2) == HIGH)) // transition condition (rising edge)  
572         {  
573             counter2++; //on transition action  
574             currentState4 = stateB4;  
575             INIT4 = 0; //resets the init indicator  
576         }  
577         lastStatebotao2 = digitalRead(botao2);  
578         delay(20); //to avoid bouncing (optional)  
579     }  
580  
581     break;  
582  
583 case stateB4:  
584  
585     if(INIT4 == 0)  
586     {  
587         Serial.println("stateB4");  
588         previousMillis4 = currentMillis;  
589         freezeSM2 = true; //on entry action  
590         INIT4=1;  
591     }  
592  
593     timerState4 = currentMillis - previousMillis4;  
594  
595     if((digitalRead(botao2) != lastStatebotao2)) // transition condition (rising edge)  
596     {  
597         if((digitalRead(botao2) == HIGH)) // transition condition (rising edge)  
598         {  
599             counter2++; //on transition action  
600             currentState4 = stateC4;  
601             INIT4 = 0; //resets the init indicator  
602         }  
603         lastStatebotao2 = digitalRead(botao2);  
604         delay(20); //to avoid bouncing (optional)  
605     }  
606  
607     break;  
608  
609 case stateC4:  
610  
611     if(INIT4 == 0)  
612     {  
613         Serial.println("stateC4");
```

```
614     previousMillis4 = currentMillis;
615     freezeSM2 = false; //on entry action
616     INIT4=1;
617 }
618
619     timerState4 = currentMillis - previousMillis4;
620
621     if((digitalRead(botao2) != lastStatebotao2)) //transition condition (
622         rising edge)
623     {
624         if((digitalRead(botao2) == HIGH)) //transition condition (rising edge)
625         {
626             counter2++; //on transition action
627             currentState4 = stateD4;
628             INIT4 = 0; //resets the init indicator
629         }
630         lastStatebotao2 = digitalRead(botao2);
631         delay(20); //to avoid bouncing (optional)
632     }
633
634     break;
635
636 case stateD4:
637
638     if(INIT4 == 0)
639     {
640         Serial.println("stateD4");
641         previousMillis4 = currentMillis;
642         deactivateSM2 = true; //on entry action
643         INIT4=1;
644     }
645
646     timerState4 = currentMillis - previousMillis4;
647
648     if((digitalRead(botao2) != lastStatebotao2)) //transition condition (
649         rising edge)
650     {
651         if((digitalRead(botao2) == HIGH)) //transition condition (rising edge)
652         {
653             counter2++; //on transition action
654             currentState4 = stateE4;
655             INIT4 = 0; //resets the init indicator
656         }
657         lastStatebotao2 = digitalRead(botao2);
658         delay(20); //to avoid bouncing (optional)
659     }
660
661     break;
```

```
661     case stateE4 :  
662  
663         if (INIT4 == 0)  
664         {  
665             Serial.println("stateE4");  
666             previousMillis4 = currentMillis;  
667             forcedCurrentState2 = stateA2;  
668             forceStateSM2 = true;  
669             INIT4=1;  
670         }  
671  
672         timerState4 = currentMillis - previousMillis4;  
673  
674         if ((digitalRead(botao2) != lastStatebotao2)) // transition condition (rising edge)  
675         {  
676             if ((digitalRead(botao2) == HIGH)) // transition condition (rising edge)  
677             {  
678                 counter2++; //on transition action  
679                 currentState4 = stateA4;  
680                 INIT4 = 0; //resets the init indicator  
681             }  
682             lastStatebotao2 = digitalRead(botao2);  
683             delay(20); //to avoid bouncing (optional)  
684         }  
685  
686         break;  
687  
688     default: break;  
689 }  
690 }
```



# Referências

- [1] Max Fizz Technologies, “What is Embedded Systems.” <https://www.maxfizz.com/what-is-embedded-systems/>. Accessed on June 10, 2019.
- [2] S. Heath, *Embedded Systems Design*. Newnes, 2003.
- [3] N. S. Kumar, M. Saravanan, and S. Jeevananthan, *Microprocessors and microcontrollers*. Oxford University Press, Inc., 2011.
- [4] P. Portugal and A. Carvalho, “The GRAFCET Specification Language,” in *The Industrial Information Technology Handbook* (R. Zurawski, ed.), ch. 64, CRC press, 2004.
- [5] M. Samek, “State Machines for Event-Driven Systems.” <https://barrgroup.com/Embedded-Systems/How-To/State-Machines-Event-Driven-Systems>, 2016. Accessed on June 10, 2019.
- [6] Tutorialspoint, “Moore and Mealy Machines.” [https://www.tutorialspoint.com/automata\\_theory/moore\\_and\\_mealy\\_machines.htm](https://www.tutorialspoint.com/automata_theory/moore_and_mealy_machines.htm). Accessed on June 10, 2019.
- [7] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.
- [8] G. Booch, J. Rumbaugh, and I. Jacobson, “Unified modeling language user guide, the (2nd edition) (addison-wesley object technology series),” *J. Database Manag.*, vol. 10, 01 1999.
- [9] M. Samek, *Practical UML Statecharts in C/C++, Event-Driven Programming for Embedded Systems*. Newnes, 2008.
- [10] Max Fizz Technologies, “Components of Embedded System - Hardware and Software Components.” <https://www.maxfizz.com/components-of-embedded-system/>. Accessed on June 10, 2019.
- [11] Max Fizz Technologies, “Characteristics of Embedded System.” <https://www.maxfizz.com/characteristics-of-embedded-system/>. Accessed on June 10, 2019.
- [12] Mehedi Hasan, “Top 15 Best Embedded Systems Programming Languages.” <https://www.ubuntupit.com/top-15-best-embedded-systems-programming-languages/>. Accessed on June 10, 2019.
- [13] Quantum Leaps, “Modern Embedded Systems Programming: Beyond the RTOS.” [http://www.state-machine.com/doc/Beyond{\\_\\_}the{\\_\\_}RTOS{\\_\\_}Notes.pdf](http://www.state-machine.com/doc/Beyond{__}the{__}RTOS{__}Notes.pdf). Accessed on June 10, 2019.

- [14] N. Jones, “Efficient C Tip #12 – Be wary of switch statements.” <https://embeddedgurus.com/stack-overflow/2010/04/efficient-c-tip-12-be-wary-of-switch-statements/>. Accessed on June 10, 2019.
- [15] Quantum Leaps, “About Quantum Leaps.” <https://www.state-machine.com/about/>. Accessed on June 10, 2019.
- [16] M. Samek, “Free state machine tool for embedded systems.” <https://embeddedgurus.com/state-space/2010/11/free-state-machine-tool-for-embedded-systems/>, 2010. Accessed on June 10, 2019.
- [17] Quantum Leaps, “Our Customer and Markets.” <https://www.state-machine.com/about/customers>, 2018. Accessed on June 10, 2019.
- [18] Quantum Leaps, “About QM™.” <https://www.state-machine.com/qm/>. Accessed on June 10, 2019.
- [19] MathWorks, “StateFlow - MATLAB & Simulink.” <https://www.mathworks.com/products/stateflow.html>. Accessed on June 10, 2019.
- [20] IAR Systems, “IAR Systems - Home Page.” <https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/visualstate/>. Accessed on June 10, 2019.
- [21] IAR Systems, “IAR Visual State - Get Started.” <https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/visualstate/video-demos/>. Accessed on June 10, 2019.
- [22] Itemis AG, “What are YAKINDU Statechart Tools.” <https://www.itemis.com/en/yakindu/state-machine/>. Accessed on June 10, 2019.
- [23] L. MKLabs Co., “StarUML 3.” <http://staruml.io/>. Accessed on June 10, 2019.
- [24] Visual Paradigm, “Ideal Modeling & Diagrammin Tool for Agile Team Collaboration.” <https://www.visual-paradigm.com/>. Accessed on June 10, 2019.
- [25] National Instruments, “What is Lab View.” <https://www.ni.com/en-us/shop/labview.html>. Accessed on June 10, 2019.
- [26] Evidence, “Sm Cube - Finite State Machine Automatic Generate Code.” <http://www.evidence.eu.com/products/smcube.html>. Accessed on June 10, 2019.
- [27] C. Rapp, “SMC: The State Machine Compiler.” <http://smc.sourceforge.net/>. Accessed on June 10, 2019.
- [28] Colm Networks, “Ragel State Machine Compiler.” <http://www.colm.net/open-source/ragel/>. Accessed on June 10, 2019.
- [29] F. Heem, “State machine generator & state diagram editor - StateForge.” <http://www.stateforge.com/>. Accessed on June 10, 2019.
- [30] B. Vogel-heuser, D. Friedrich, U. Katzke, and D. Witsch, “Usability and benefits of UML for plant automation – some research results,” vol. 3, no. 1, pp. 52–60, 2005.

- [31] J. Ferreira, “Ambiente integrado em eclipse para desenvolvimento de programas iec 61131-3,” Master’s thesis, FEUP, 2015.
- [32] J. Antunes, “Desenvolvimento e integração de editores gráficos de elevado impacto visual,” Master’s thesis, FEUP, 2010.
- [33] J. Peixoto, “Multi-touch interaction for interface prototyping,” Master’s thesis, FEUP, 2013.
- [34] Microsoft, “Choose your app platform.” <https://docs.microsoft.com/en-us/windows/apps/desktop/choose-your-platform>, 2019. Accessed on June 10, 2019.
- [35] Microsoft, “Writing a T4 Text Template.” <https://docs.microsoft.com/en-us/visualstudio/modeling/writing-a-t4-text-template?view=vs-2019>. Accessed on June 10, 2019.
- [36] R. Gonçalves, “UPS portátil,” Master’s thesis, FEUP, "A Publicar", 2019.