UNIVERSIDADE FEDERAL DE PERNAMBUCO

Universidade Federal de Pernambuco
Centro de Tecnologia e Geociências
Departamento de Eletrônica e Sistemas

Graduação em Engenharia Eletrônica

Angélica Muniz Xavier da Silva

Um compilador para geração automática de código para máquinas de estados hierárquicas com aplicações em sistemas embarcados

Recife
2023

Angélica Muniz Xavier da Silva

Um compilador para geração automática de código para máquinas de estados hierárquicas com aplicações em sistemas embarcados

Trabalho de Conclusão apresentado ao Curso de Graduação em Engenharia Eletrônica, do Departamento de Eletrônica e Sistemas, da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia Eletrônica.

Orientador(a): Prof. Hermano Andrade Cabral, Ph.D.

Recife
2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Angélica Muniz Xavier da .
Um compilador para geração automática de código para máquinas de estados hierárquicas com aplicações em sistemas embarcados / Angélica Muniz Xavier da Silva. - Recife, 2023.
114 p. : il., tab.

Orientador(a): Hermano Andrade Cabral
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia Eletrônica - Bacharelado, 2023.
Inclui referências.

1. Sistemas embarcados. 2. Máquinas de estados hierárquicas. 3. Linguagem de domínio específico. I. Cabral, Hermano Andrade. (Orientação). II. Título.

620 CDD (22.ed.)

Angélica Muniz Xavier da Silva

Um compilador para geração automática de código para máquinas de estados hierárquicas com aplicações em sistemas embarcados

Trabalho de Conclusão apresentado ao Curso de Graduação em Engenharia Eletrônica, do Departamento de Eletrônica e Sistemas, da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia Eletrônica.

Aprovado em: 02/10/2023

Banca Examinadora

Documento assinado digitalmente
gov.br
HERMANO ANDRADE CABRAL
Data: 21/08/2024 17:19:57-0300
Verifique em https://validar.iti.gov.br

Prof. Hermano Andrade Cabral, Ph.D.
Universidade Federal de Pernambuco

Documento assinado digitalmente
gov.br
GUILHERME NUNES MELO
Data: 22/08/2024 16:37:01-0300
Verifique em https://validar.iti.gov.br

Prof. Guilherme Nunes Melo, D.Sc.
Universidade Federal de Pernambuco

Documento assinado digitalmente
gov.br
DANIEL DE FILGUEIRAS GOMES
Data: 21/08/2024 12:30:44-0300
Verifique em https://validar.iti.gov.br

Prof. Daniel de Filgueiras Gomes, D.Sc.
Universidade Federal de Pernambuco

Dedico este trabalho à
minha mãe, com profundo
agradecimento e admiração.

Agradecimentos

Primeiramente, gostaria de agradecer à minha família pelo amor e suporte incondicionais que me proporcionaram durante todos esses anos de estudos, especialmente minha mãe e minha tia Ângela.

Também gostaria de agradecer a meus amigos, pois sempre estivemos juntos, nos motivando e nos encorajando, mesmo nos momentos mais desafiadores. Suas palavras de incentivo e amizade sincera me mantiveram motivada e me ajudaram a superar as dificuldades.

Por fim, gostaria de expressar minha gratidão ao meu orientador, que me guiou e apoiou durante todo o processo de elaboração deste trabalho. Suas orientações e sugestões foram cruciais para a minha formação acadêmica e profissional, e sou grata por toda a dedicação e paciência durante todos esses meses.

A todos que me ajudaram neste caminho, o meu sincero agradecimento. Este trabalho é o resultado do esforço e dedicação, e sem o apoio de vocês, não seria possível alcançar este objetivo. Muito obrigada!

Os ingredientes da paz são

o pão e o amor.

Josué de Castro (1908 - 1973)

Resumo do Trabalho de Conclusão de Curso apresentado ao Departamento de Eletrônica e Sistemas, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Eletrônica(Eng.)

Um compilador para geração automática de código para máquinas de estados hierárquicas com aplicações em sistemas embarcados

Angélica Muniz Xavier da Silva

Apesar da crescente importância de sistemas embarcados em diversas áreas do conhecimento humano, o desenvolvimento de *software* robusto e confiável para estas plataformas ainda é desafiador. Devido a muitos desses sistemas serem reativos a eventos, o uso de máquinas de estado é uma prática comum. Este trabalho teve como objetivo, portanto, desenvolver uma ferramenta de *software* para, a partir de uma descrição textual de uma máquina de estados hierárquica, gerar o código C necessário para sua execução em reação a uma sequência potencialmente infinita de eventos. Para a descrição da máquina de estados, a gramática de uma linguagem descritiva simples, mas eficaz, é proposta neste trabalho. Além disso, foi desenvolvido um compilador baseado nesta linguagem para geração do código C da máquina de estados, de modo a permitir ao programador se preocupar apenas em desenvolver as ações associadas às transições da máquina. Por último, foi desenvolvida uma máquina de estados para um semáforo inteligente, a qual foi implementada em *hardware* usando o microcontrolador ATmega328p, demonstrando a utilidade da ferramenta proposta.

Palavras-chave: Sistemas embarcados; máquinas de estados hierárquicas; linguagem de domínio específico.

Abstract of Course Conclusion Work, presented to Departament of Eletronic and Systems, as a partial fulfillment of the requirements for the degree of Bachelor of Electronic Engineering(Eng.)

A Compiler for Automatic Code Generation for Hierarchical State Machines with Applications in Embedded Systems

Angélica Muniz Xavier da Silva

Despite the growing importance of embedded systems in various areas of human knowledge, the development of robust and reliable software for these platforms remains a challenge. As many of these systems are reactive to events, the use of state machines is a common practice. Therefore, the goal of this work was to develop a software tool that, based on a textual description of a hierarchical state machine, generates the necessary C code for its execution in response to a potentially infinite sequence of events. For the state machine description, the grammar of a simple yet effective descriptive language is proposed in this work. Additionally, a compiler based on this language was developed to generate the C code for the state machine, allowing the programmer to focus on developing the actions associated with the transitions of the machine. Finally, a state machine for an intelligent traffic light was developed, which was implemented in hardware using the ATmega328p microcontroller, demonstrating the utility of the proposed tool.

Keywords: Embedded Systems; hierarchical state machines; domain-specific language.

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

kW	... Quilowatt
MHz	Milhão de Hertz
V	..... Volt

Lista de Abreviações

ENIAC	Electronic Numerical Integrator and Computer
IBM	International Business Machines Corporation
ARM	Advanced RISC Machine
AVR	Advanced Virtual RISC
ABS	Anti-lock braking system
CPU	Central Processing Unit
DSL	Domain Specific Language
UML	Unified Modeling Language
USB	Universal Serial Bus
NASA	National Aeronautics and Space Administration
AST	Abstract Syntax Trees
EBNF	Extended Backus-Naur form
BSP	Board Support Package
PCB	Printed circuit board
LED	Light-emitting diode
DIP	Dual In-line Package
SMD	Surface Mounted Devices
DC	Direct current
TVS	Transient-voltage-suppression diode
SPI	Serial peripheral interface
OMG	Object Management Group

Sumário

Capítulo 1

Introdução

1.1 Motivação

Sistemas embarcados tornaram-se um grande expoente no desenvolvimento tecnológico de todas as áreas do conhecimento humano nas últimas décadas. É visível sua importância na configuração de muitos dos sistemas da atualidade, em diversos setores como a saúde, transporte, segurança, geração e distribuição de energia, comunicações e entretenimento (LEE, 2011).

Diante da larga escala em que os sistemas embarcados vêm sendo empregados, estão se tornando cada vez mais baratos e acessíveis. Os esforços direcionados à melhoria dos elementos desse sistema também resultaram em avanços quanto ao consumo de energia, diminuição de tamanho e processamentos mais velozes de seus componentes, aumentando ainda mais suas aplicações.

Entretanto, com o avanço no *hardware* desses sistemas embarcados, sua programação vem ficando cada vez mais complexa. Ferramentas que ajudem a controlar essa complexidade se tornam, portanto, cada vez mais importantes.

Como a operação de muitos sistemas embarcados é predominantemente de reação a eventos externos e internos, o paradigma de programação orientada a eventos pode trazer benefícios no entendimento do problema e implementação da solução (PICARD, 2018).

Em muitos desses casos, a aplicação tem um estado que depende dos eventos

passados e que influencia as ações futuras do sistema. Assim, a técnica de programação por máquina de estados (SAMEK, 2008) é particularmente interessante no desenvolvimento desses sistemas, sendo o foco deste trabalho.

1.1.1 Importância de sistemas embarcados

O primeiro computador eletrônico produzido, o ENIAC, iniciando então a primeira geração de computadores na década de 50, ocupava uma área de 72 metros quadrados e pesava em torno de 30 toneladas. Destinado a resolver apenas problemas específicos, consumia em torno de 200kW, o que implicava na necessidade de um sistema de refrigeração à altura. Para resolver um problema diferente, era necessário a uma reprogramação completa (IFRAH, 2000).

O desempenho do ENIAC era notável para a sua época, uma vez que atingia velocidades até então inigualáveis para realizar operações. Variando entre 200 microssegundos até 6 milissegundos, a depender da quantidade de dígitos envolvidos na adição, multiplicação e até mesmo divisão.

Cada painel de controle possuía 1.456 botões de discagem e podia armazenar uma quantidade significativa de informações. No entanto, as configurações tinham que ser feitas manualmente, inserindo cada dígito de instrução individualmente. Para iniciar a máquina, o operador precisava ajustar um total de 4.368 interruptores nos três painéis. Quaisquer procedimentos adicionais exigiam instruções detalhadas e ajustes, antes que a máquina pudesse ser iniciada.

Tamanha a preparação para iniciar a máquina, que erros humanos eram comuns, o que tornavam os cálculos mais imprecisos. Porém, mesmo removendo o fator do erro humano, os próprios inventores do ENIAC admitiram que a máquina obteve o resultado correto apenas 20 vezes em 100.

Com diversos interruptores e luzes de exibição, o ENIAC era também composto por 10 mil condensadores, seis mil interruptores, 1,5 mil relés, 50 mil resistores e mais 500 mil juntas de solda. Além disso, possuía 18 mil válvulas, responsável por uma porcentagem considerável de seu consumo (IFRAH, 2000).

Diante da necessidade da substituição das válvulas por componentes que apresentassem um consumo menor e uma durabilidade maior, os transistores passaram a ser integrados aos painéis dos computadores tão logo foram desenvolvidos, em 1947, pelos físicos William Shockley, John Bardeen e Walter Brattain, que integravam a empresa Bell Laboratories. A matéria prima deste componente semicondutor nada mais é do que o Silício, abundantemente encontrado no planeta, o que o torna mais viável economicamente e possibilita um produto final com dimensões reduzidas em comparação com a válvula elétrica (BOYLESTAD, 2012).

O uso dos transistores marcou a segunda geração das máquinas de computar. A economia tanto no consumo para 150kW quanto no custo de fabricação, o aumento da frequência da execução de tarefas, o *clock*, para 1MHz, a diminuição das dimensões do computador em torno de 100 vezes, resolveu não só o problema do número demasiado de manutenções, como também trouxe uma série de vantagens em relação ao uso da válvula elétrica. Além disso, a linguagem de programação *Assembly* passou a ser a utilizada para programar o computador, substituindo as linguagens de máquina usadas anteriormente (TORRES, 2017).

Por exemplo, o *mainframe* 7090 da IBM era um sucessor direto transistorizado do 709, baseado em válvulas, e tinha uma velocidade de processamento cinco vezes maior do que o seu antecessor, melhor aproveitamento do espaço devido ao seu menor tamanho, e custava menos para instalar e operar do que os 709 anteriores (IBM, 2023).

Vale ressaltar então a importância do material semicondutor na indústria eletrônica e seu grande potencial recém-descoberto, que proporcionou aumentos significativos na velocidade, na eficiência e na miniaturização, não só dos componentes como também dos aparelhos.

Em seguida, entre os anos de 1965 e 1975, uma segunda revolução ocorreu graças à compactação de arranjos de transistores em circuitos integrados, surgindo então a terceira geração de computadores (IFRAH, 2005). Mais uma vez houve diminuição do consumo e custo de fabricação, além da integração com teclados e monitores,

que permitiam a visualização dos sistemas operacionais ainda bastante primitivos. Dessa vez, os computadores tornaram-se mais acessíveis e com a possibilidade do incremento de sua capacidade, de acordo com a necessidade do usuário, pagando relativamente pouco.

O surgimento e uso dos processadores e microprocessadores assinalou o início da geração de computadores atual, a quarta, em 1975. Houve então a disseminação dos microcomputadores que ofereceram uma enorme gama de novas opções para os usuários (TORRES, 2017).

Além dos microcomputadores, que podem ser utilizados para a realização de diversas tarefas, surgiram os sistemas embarcados, estes que também são dispositivos microprocessados ou microcontrolados porém com a finalidade de realizar tarefas específicas. Por exemplo, chips que leem os sensores de temperatura de ares-condicionados e sinais do controle remoto para gerir o funcionamento de um aparelho (LEE, 2011).

Há vantagens em utilizar microcontroladores em sistemas embarcados, uma vez que já possuem alguns periféricos integrados no mesmo chip e há uma variedade de arquiteturas disponíveis, tais como ARM, x86, Harvard, Atmel AVR, entre outras (WIKIPEDIA, 2023).

Desde então, os sistemas embarcados foram amplamente utilizados em quase todos os setores, impulsionando o desenvolvimento tecnológico de todas as áreas do conhecimento humano. Possui aplicações na saúde, como em marcapassos; no transporte, como em freios ABS; na telecomunicação, como em satélites; no lazer, como em televisores inteligentes.

Vale ressaltar que evoluções tecnológicas significativas, como a Internet das Coisas e a Quarta Revolução Industrial, têm o sistema embarcado como um de seus pilares fundamentais. Dessa forma, fica claro a importância dos dispositivos microcontrolados para diversas economias e como impulsionadores do desenvolvimento tecnológico (WIKIPEDIA, 2021).

As características de sistemas embarcados serão explanadas a seguir, proporcio-

nando um melhor entendimento acerca desse importante tema.

1.1.2 Características de sistemas embarcados

Diante da diversidade dos eletrônicos, os sistemas embarcados destacam-se por sua habilidade computacional. Entretanto, comparado a um computador convencional, esses apresentam tamanho e capacidade operacional reduzidas (EMBARCADOS, 2023), o que não necessariamente são desvantagens. Claramente, o tamanho reduzido é uma vantagem por oferecer a possibilidade de embutir o sistema e a maioria de seus periféricos, dentro do aparelho eletrônico, sem ocupar uma quantidade considerável de espaço. Já a capacidade de processamento reduzida não apresenta perdas para os projetos dos sistemas embarcados, pois a capacidade que possuem é suficiente para uma operação eficiente, para a maioria das aplicações (JIMÉNEZ ROGELIO PALOMERA, 2014).

Em virtude da necessidade de lidar com tarefas reduzidas e específicas, os sistemas embarcados operam bem sem um investimento elevado em memória, núcleos ou velocidade de *clock* alta. Como resultado, apresentam custo reduzido, o que é mais uma vantagem para sua utilização. Em adição à característica de operar com uma quantidade restrita de tarefas, a chance de apresentar erros ou falhas durante a operação é menor, tornando a eficiência do sistema mais um ponto positivo. Dessa maneira, frente à sua importância em áreas críticas como a saúde e o transporte, o fator eficiência é de grande destaque.

A estrutura geral de um sistema embarcado pode ser separada em dois grandes componentes: um conjunto de *hardware*, que inclui o microprocessador; e uma série de programas de *software*, formando o *firmware* do sistema. À vista disso, o *firmware* é o que dá a funcionalidade ao *hardware*.

Aprofundando um pouco mais no *hardware* dos sistemas eletrônicos microcontrolados, é a partir desse importante componente, o *hardware*, que o sistema se torna capaz de interagir com o meio. Ou seja, o microcontrolador se comunica com diversos periféricos, sejam eles sensores e/ou atuadores. Dos sensores se obtém da-

dos relativos ao ambiente externo, que serão processados e, de acordo com estes, o sistema poderá reagir no ambiente, controlando os atuadores (AMOS, 2020).

Eventos internos ao microcontrolador também influenciam na operação do sistema embarcado, como exemplo um temporizador, que pode ser configurado para acionar uma tarefa ou alterar o valor de uma variável (LEE, 2011).

Reagir a eventos internos e externos de forma eficiente caracteriza a "programação reativa" utilizada nesses sistemas, citada neste parágrafo de forma introdutória e explanada adiante. Por consequência, a programação do sistema embarcado é de fundamental importância para a sua eficiência e confiabilidade. Saber aproveitar seus recursos limitados e desenvolver um programa que não apresente erros e minimize a necessidade de manutenção é crucial. Com o avanço da microeletrônica e o consequente aumento no desempenho dos microcontroladores, essa questão da programação tende a ficar cada vez mais importante, assim como o desenvolvimento e uso de sistemas operacionais mais simples, voltados para sistemas embarcados (AMOS, 2020).

Em geral, sistemas embarcados baseados em microcontroladores, principalmente os mais simples, não utilizam sistemas operacionais, e a maneira mais comum de programá-los é a partir de um *superloop*, ou seja, um grande laço principal, executado de maneira contínua e sequencial. Entretanto, o *superloop* não é indicado para programas complexos, no qual há a possibilidade de ocorrer alguns problemas, como conflitos entre diversos níveis de interrupção (SAMEK, 2012).

Isso é particularmente verdade para muitos sistemas embarcados, que operam reagindo a eventos externos e internos. Nestes casos, uma programação do tipo orientada a eventos é uma forma eficiente e objetiva de se implementar o sistema. Além disso, no tratamento dos eventos, muitas vezes é interessante usar o paradigma de programação reativa envolvendo sequências de dados e propagação de mudanças. Estes dois conceitos estão explicados na seção a seguir.

Todavia, com o avanço da microeletrônica, os microcontroladores tornaram-se mais robustos, possibilitando o desenvolvimento de sistemas operacionais mais sim-

ples, voltados para sistemas embarcados.

1.1.3 Programação orientada a eventos e programação reativa

Dentre os vários paradigmas de programação, como procedural, funcional e orientada a objetos, dois se destacam no contexto de sistemas embarcados: programação orientada a eventos e programação reativa. Os motivos são descritos a seguir.

Programação orientada a eventos é um paradigma baseado no conceito que o estado de operação normal da CPU é um estado dormente, onde nada de útil é feito. A CPU só sai deste estado quando um evento ocorre e precisa ser tratado. Neste momento, a função correta para aquele evento, é identificada e executada, ao término da qual a CPU volta ao estado dormente (SAMEK, 2003). Este paradigma então se preocupa em projetar e implementar as várias rotinas de tratamento de eventos para que a programação e execução sejam feitas da forma mais eficiente e clara possível.

No decorrer da execução da função de tratamento de eventos, pode ser interessante utilizar o paradigma de programação reativa. Este paradigma se preocupa com o gerenciamento das mudanças de estado de qualquer entidade que é afetada pela ocorrência do evento, e tem sido sugerido como uma solução apropriada para o desenvolvimento de sistemas voltados a eventos (BAINOMUGISHA et al., 2013).

A popularidade da programação reativa só aumenta em virtude de seus benefícios, como a criação de aplicações altamente disponíveis e de curto tempo de resposta. Estes são benefícios que se aplicam não só ao desenvolvimento *web*, mas também ao desenvolvimento de programas embarcados.

Segundo (PICARD, 2018), a forma de programação tradicional, conhecida como a programação procedural ou estruturada, utiliza uma lógica de tarefas síncronas, ou seja, que se comunicam em tempos determinados, seguindo uma ordem cronológica e linear. A desvantagem dessa maneira de programar mais rígida é a dificuldade de lidar com falhas ou com demandas não previstas.

Um dos pilares da programação reativa é o gerenciamento de eventos assíncronos, pois, diante da natureza dos eventos, estes podem ocorrer a qualquer momento. Um fator positivo desse gerenciamento é o fato de poder estabelecer prioridades entre os eventos, reagindo assim de acordo com a importância.

Essa abordagem acaba trazendo, naturalmente, vantagens como uma utilização mais eficiente dos recursos computacionais, a diminuição da latência e de erros que possam surgir e travar o processamento. O sistema, então, torna-se mais inteligente, pois tem a capacidade de gerenciar falhas, reagir a demandas não previstas, seguir caminhos alternativos e funcionar em tempo real (PICARD, 2018).

Entretanto, a desvantagem que se pode destacar dessa forma de programação é a curva de aprendizado. Desenvolver um programa baseado em eventos assíncronos não é a maneira mais comum que a maioria das pessoas estão habituadas, portanto pode ser complicado no começo, momento no qual os conceitos desta programação não estão plenamente absorvidos. Além disso, a depuração pode se mostrar confusa, uma vez que as operações não são executadas sequencialmente, mas sim de maneira assíncrona e simultânea.

O desafio continua quando se deseja escrever um código legível, para que outras pessoas o possam reutilizar, testar e fazer as manutenções necessárias. Um programa orientado a eventos, em suma, é mais difícil de ser implementado e lido, por não tomar um rumo sequencial habitual da leitura de seres humanos. Uma ação ocorre após um evento, e as ações seguintes apenas dependem dos próximos eventos, que deverão ser tratados. Dessa forma, a complexidade do programa aumenta, e torna-se um trabalho árduo entender o que está acontecendo (PICARD, 2018).

Em virtude da complexidade com que a programação assíncrona lida, a utilização de máquina de estados como ferramenta de suporte, é primordial para que o sistema não se perca no fluxo de dados e possa tomar as decisões corretamente, garantindo, assim, o bom funcionamento do sistema eletrônico. Dessa maneira, o tema da próxima seção é a programação com máquina de estados.

1.1.4 Programação por máquina de estados

A descrição do comportamento de sistemas embarcados pode ser extremamente difícil, devido à complexidade crescente dos aparelhos eletrônicos. Naturalmente, no início de qualquer projeto, não há a completa compreensão de todas as ações do sistema; como resultado, poderão surgir vários erros de implementação, devido às descrições incompletas, que não compreendem todas as possibilidades. Dessa forma, a utilização de modelos ajuda na captura e descrição do comportamento de maneira precisa (VAHID, 2001).

Dentre os modelos de computação mais comuns, como os modelos de programação sequencial e o de processos comunicantes, o modelo de máquina de estados é mais indicado para sistemas de controle e monitoramento de entradas e saídas. Há diversas linguagens de programação que podem capturar este modelo; a saber linguagens como C, C++, Java, Python, entre outras.

Em geral, um modelo de máquina de estados define que um sistema contém uma quantidade de estados possíveis, em que cada um representa uma situação relevante do sistema. Ademais, apenas um estado por vez é permitido, porém, a transição de um estado para outro é possível, assim que determinada condição seja satisfeita. Estas transições ajudam a definir o fluxo e funcionamento do *software* em questão, além das ações que podem ser acionadas no decorrer deste fluxo. A complexidade do mecanismo do fluxo da máquina de estado é definida por seu desenvolvedor e pode ser tão grande quanto a complexidade do sistema a ser descrito (VAHID, 2001).

Espera-se de um sistema embarcado que mantenha-se em operação por um longo período de tempo, ininterruptamente, e que possa, inclusive, recuperar-se da maioria dos erros sozinho. Dessa maneira, a programação por máquina de estados oferece o suporte necessário para esta atividade, uma vez que, a sua utilização proporciona um bom direcionamento do funcionamento do sistema.

No entanto, programar uma máquina de estados pode ser extremamente entendiante e suscetível a erros, como realizar transições para estados errôneos ou mesmo suprimir ações referentes às transições. Haja vista que a forma mais fácil de desen-

volver uma máquina de estados é a partir de sua representação gráfica, o trabalho em questão se propõe a especificar uma máquina de estados de uma forma programática a partir de uma descrição textual obtida facilmente de uma representação gráfica. Assim, este trabalho é fundamental para garantir maior robustez e facilidade ao programar sistemas complexos (TORRES, 2017).

A programação por máquina de estados desenvolvida neste trabalho apresenta ainda a possibilidade de ser implementada junto a um sistema operacional de tempo real. A vantagem oferecida por este tipo de configuração é a de uma programação com um nível de abstração suficiente de modo que o mesmo programa seja utilizado em diferentes chips, ou seja, facilitando a adaptação do mesmo *firmware* a outros *hardwares* embarcados mais evoluídos ou com arquiteturas dessemelhantes. A desvantagem dessa estratégia, no entanto, é o aumento do consumo de memória e de processamento.

1.2 Objetivos

1.2.1 Objetivos Gerais

O objetivo deste trabalho consiste em desenvolver um compilador para definição programática de uma máquina de estados a partir de uma descrição textual.

1.2.2 Objetivos Específicos

Os objetivos específicos consistem em:

- Definir a descrição textual da máquina de estados.
- Definir a gramática da linguagem, formalizando essa descrição.
- Desenvolver o compilador que gerará o código C, que represente a máquina de estados a partir da descrição textual, baseada na gramática elaborada.

1.3 Estrutura do Trabalho

Este trabalho é dividido em capítulos que vão desde a introdução de conceitos essenciais para o entendimento do projeto, passando por sua implementação até a sua aplicação em um exemplo de sistema embarcado. Em resumo, os capítulos dividem-se em:

Capítulo 2 - Fundamentação: apresenta a fundamentação teórica necessária para o projeto. Esse capítulo abrange os conceitos de Programação Reativa e Orientada a Eventos, Máquina de Estados e Linguagem de Domínio Específico.

Capítulo 3 - Métodos e Materiais: com os conceitos já explanados, este capítulo explicita não só o método seguido para desenvolvimento do trabalho, como também as ferramentas utilizadas para implementação deste. A saber, a linguagem de programação Python e sua biblioteca Lark.

Capítulo 4 - Implementação: a partir dos conceitos e ferramentas apresentados, demonstra-se como foi executado o projeto. Esse capítulo explicita a Gramática desenvolvida, apresenta a Geração do Código e Estrutura do Programa Gerado.

Capítulo 5 - Resultados: aborda a aplicação do presente projeto em um sistema embarcado.

Capítulo 6 - Conclusão: apresenta as conclusões do trabalho, como também sugestões de trabalhos futuros.

Capítulo 2

Fundamentação

Neste capítulo serão apresentados alguns conceitos que foram aplicados no desenvolvimento deste TCC. Será introduzido inicialmente o conceito de programação reativa e orientada a eventos, importantes paradigmas no âmbito de sistemas embarcados. Em seguida, será definido o que é uma máquina de estados, um poderoso conceito para ser utilizado com programação reativa. Será descrito o principal tipo de máquina de estados a ser considerado neste trabalho, a hierárquica. Por último, será apresentado o conceito de Linguagem de Domínio Específico, ou DSL (do inglês *Domain Specific Language*), a ser usado para simplificar a descrição e programação da máquina de estados.

2.1 Programação reativa

No contexto da programação, a reatividade é definida como o processo de receber estímulos e propagar eventos, e normalmente utilizada para sistemas que respondem a temporizadores, sensores ou outros periféricos. Esta programação é orientada a fluxos de dados e propagações de estados, nos quais, estes fluxos, são em grande parte assíncronos. Desta maneira, as operações não são, necessariamente, executadas de forma sequencial. Salienta-se que, ao observar o fluxo de um programa reativo, percebe-se que todas as ações tomadas são formadas a partir de reações a eventos, mensagens, chamadas e falhas (GUEDES, 2021).

Um dos fatores que contribuem para a melhor eficiência na utilização dos recursos computacionais e menor latência desta linguagem, é o uso de processos não-bloqueantes. Assim sendo, a execução das tarefas dificilmente terá a tendência de criar bloqueio ou sobrecarga da memória e do processador. Dentre os vários modos de criação de processos não-bloqueantes, pode-se citar tanto o emprego de sub-rotinas, como o de mensagens de erros que são emitidas pelo próprio fluxo, permitindo que o tratamento destes seja mais eficiente.

Em 2014, um grupo de desenvolvedores criou o documento Manifesto Reativo que descreve os quatro pilares que programas devem ter para serem considerados reativos (BONÉR et al., 2014). Dentre eles, os que se aplicam aos sistemas embarcados são explanados a seguir.

O pilar da resiliência define que um programa reativo deve ser capaz de se recuperar e de lidar com falhas, uma vez que, durante a operação, é comum ocorrer falhas originadas do *software*, do *hardware* ou até mesmo da conectividade. Este é um importante aspecto para que o sistema seja responsivo após uma falha. A citar dois mecanismos que podem ser empregados nesta tarefa, há o mecanismo de isolamento e o de delegação.

O mecanismo de isolamento pode ser definido como o desacoplamento do tempo e do espaço. Desacoplar no tempo significa que o transmissor e o receptor não precisam estar disponíveis ao mesmo tempo para que a comunicação seja realizada, devendo ser efetuada então a partir de troca de mensagens de modo assíncrono. Já o desacoplamento no espaço, define que o transmissor e o receptor não precisam ser executados no mesmo processo, mas sim quando o núcleo do sistema decidir, o que torna o programa mais eficiente. Em virtude dos benefícios deste mecanismo, é possível que as falhas sejam capturadas, sinalizadas e gerenciadas de maneira proveitosa, diferentemente do que se fosse corrigida de maneira sequencial.

Outro mecanismo que vale a pena ressaltar é o de delegação assíncrona, definido como quando um componente delega a execução de uma tarefa a outro componente, deixando-se livre para executar outra tarefa ou até mesmo apenas para observar, e

assim ficar mais responsivo às falhas ou outros estímulos (BONÉR et al., 2014).

Ser guiado por mensagens é mais um pilar definido pelo Manifesto Reativo, e é a base de programas reativos. As mensagens são trocadas de maneira assíncrona e coordenadas por gerenciadores de eventos assíncronos e não-bloqueantes. Ademais, este pilar reforça os mecanismos de desacoplamento e de delegação.

Uma importante característica do programa reativo é reagir automaticamente a mudanças nos dados de entrada, propagando essas alterações ao longo do fluxo de dados. Isso permite construir sistemas que são resilientes, responsivos e flexíveis. Essa característica oferece a vantagem de que os programadores não precisam se preocupar com a ordem dos eventos e dependências de computação (BAINOMUGISHA et al., 2013).

Como resumo, para elucidar melhor o paradigma da programação reativa, (BAINOMUGISHA et al., 2013) faz um analogia com planilhas e suas células de dados e fórmulas. Em uma planilha, as células contêm valores ou fórmulas, e quando o valor de uma célula é alterado, as fórmulas relacionadas são recalculadas automaticamente. A programação reativa consiste em incorporar o modelo de planilha em linguagens de programação, permitindo a atualização automática e dinâmica de dados com base em suas dependências. Ou seja, fica explícito o benefício da capacidade de gerenciar automaticamente as dependências de dados.

Como mencionado em 1.1.3, o paradigma da programação reativa é adequado para o desenvolvimento de aplicações orientadas a eventos, objeto da próxima seção.

2.2 Programação orientada a eventos

Por volta da década de 1960, com a introdução do mouse e teclado, foi necessário que os programadores pensassem em termos de programação orientada a eventos. Uma vez que os eventos poderiam se originar de várias fontes, como os agendados, advindos do *hardware*, do próprio sistema operacional, ou outros eventos específicos do domínio do problema, foi necessário a criação de ferramentas para manipulação dos eventos e especificação de ações associadas, uma vez que o fluxo do programa

agora é definido pelos eventos, e não mais pelo programador (YEAGER, 2014).

Apesar de existir desde o início da ciência da computação, a programação orientada a eventos ainda é difícil de ser utilizada corretamente. O desafio vai além de escrever código orientado a eventos e envolve torná-lo legível, fácil de realizar a manutenção, reutilizável e testável. A programação orientada a eventos é mais complexa de implementar e entender do que a programação sequencial, pois envolve escrever código que nem sempre é fácil de ser compreendido pelos seres humanos. Ao invés de uma sequência linear de ações, as ações nesse tipo de programação são desencadeadas por eventos e podem estar dispersas no programa. Quando o fluxo de código se torna complexo, é difícil acompanhar o que está acontecendo (PICARD, 2018).

O programa precisa ser desenvolvido para ser adaptável conforme a ocorrência de eventos, uma vez que não é mais necessário que o programador gerencie uma única sequência de execução do início ao fim. Agora, o programador deve se concentrar nas ações que devem ser realizadas quando um determinado evento ocorrer. (YEAGER, 2014).

Vale ressaltar que essa visão da execução do programa foi inspirada nas interrupções de *hardware* e seus manipuladores. A ideia de lidar com eventos de forma reativa e adaptável encontra paralelos com a forma como o *hardware* lida com interrupções, onde a execução normal do programa é temporariamente suspensa para lidar com um evento específico antes de retomar o fluxo de execução normal.

Especificando o que é evento, pode-se dizer que durante a execução de um programa, um evento ocorre quando uma determinada condição é atingida, exigindo a realização de ações específicas. Nesse contexto, um manipulador de eventos, ou *event handler* em inglês, é uma parte ativa do programa que é acionada em resposta a um evento. Adicionalmente, vale ressaltar que é criada uma fila de eventos para armazenar os que já ocorreram, mas que ainda não foram processados pelo manipulador. Dessa forma, o primeiro da fila, quando chamado pelo manipulador de eventos, é removido da fila e tem suas ações desencadeadas (YEAGER, 2014).

Vale ressaltar que a programação orientada a eventos e as máquinas de estados podem trabalhar juntas para criar sistemas reativos e adaptáveis, onde o comportamento do programa é determinado pela interação com eventos externos. A máquina de estados colabora fornecendo uma estrutura conceitual para modelar o comportamento do sistema, enquanto a programação orientada a eventos fornece um mecanismo para lidar com a ocorrência desses eventos e desencadear as ações apropriadas (SAMEK, 2008). Dessa forma, a próxima seção abordará o tema Máquina de Estados com mais profundidade.

2.3 Máquina de estados

Máquinas de estados são modelos que podem ser utilizados em sistemas embarcados para descrever seu comportamento de maneira bastante precisa. Ademais, se apresentam como uma escolha apropriada para esses sistemas por possuir uma baixa necessidade de memória, uma vez que a limitação de memória tanto é comum quanto crítica, principalmente para sistemas complexos (SIPSER, 2013).

Um exemplo de dispositivo que pode ter seu funcionamento modelado por uma máquina de estados é o de um controlador de uma lâmpada de banheiro de avião. A função desta lâmpada é indicar se o banheiro está ocupado ou não. Esta lâmpada terá apenas dois estados possíveis, `Ligada` ou `Desligada`. Supondo que este banheiro tenha trava tanto do lado de dentro, quanto do lado de fora, as quatro entradas possíveis para este sistema seriam: `Acionamento da trava de dentro`, `Acionamento da trava de fora`, `Acionamento de ambas` e `Nenhuma`. Na Tabela 2.1, o número `0` sinaliza que a respectiva trava indicada no título da coluna está desativada, e o número `1` sinaliza que está ativada.

Como mostrado no diagrama de estados para esta lâmpada, representado pela Figura 2.1, o controlador muda de estado de acordo com a entrada recebida. Uma descrição de seu funcionamento seria: quando a lâmpada encontra-se no estado de `Desligada`, e recebe as entradas de `Acionamento da trava de fora` ou então `Nenhuma`, ou seja, caso a trava de fora do banheiro seja ativada ou nenhuma trava

Acion. trava de dentro	Acion. trava de fora	Entrada da máquina de estados
0	0	Nenhuma
0	1	Acion. trava de fora
1	0	Acion. trava de dentro
1	1	Acion. de ambas

Tabela 2.1: Possíveis entradas para o sistema de controle da lâmpada

Fonte: A autora.

seja acionada, a lâmpada permanece no estado de `Desligada`. Porém, caso a `Trava de dentro`, ou `Ambas` as travas sejam acionadas, a lâmpada vai para o estado de `Ligada`. Ao receber a entrada de `Nenhuma` ou `Acionamento da trava de fora`, a lâmpada muda de estado para `Desligada` novamente.

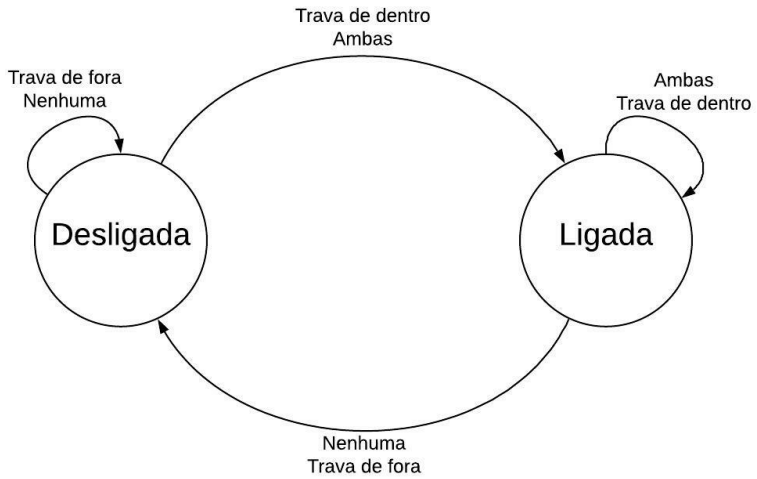

Figura 2.1: Diagrama de estados para o sistema de controle da lâmpada

Fonte: A autora.

Por apresentar dois estados possíveis, `Ligada` ou `Desligada`, apenas um *bit* de memória seria necessário para armazenar o estado da lâmpada. Em outros sistemas, como o de um elevador ou calculadora, mais estados são possíveis e mais memória será requerida de um controlador com memória finita para armazenar tanto o estado do sistema quanto as possíveis entradas. Desta forma, para uma boa performance é importante uma programação enxuta e eficaz.

Para uma definição formal de máquina de estados finita, Sipser (2013) define

uma lista de 5 objetos: um conjunto de estados; um alfabeto de entrada, também chamado de eventos; regras de movimento, também chamadas de funções de transição que modificam o estado atual da máquina de acordo com a entrada; um estado inicial; e um conjunto de estados aceitáveis, em que definem se uma máquina aceita ou não determinada sequência de eventos, ou seja, após processar todos os eventos da sequência de entrada, o último estado deve ser um estado aceitável, ou então a máquina rejeita tal sequência de eventos.

Em suma, uma máquina de estados finita é uma lista de 5 objetos, $\{Q, \Sigma, \delta, q_0, F\}$, definidos como se segue:

- Q é um conjunto finito, chamado de **estados**;
- Σ é um conjunto finito, chamado de **alfabeto** ou de **eventos**;
- δ: $Q \times \Sigma \rightarrow Q$ é a **função de transição de estado**;
- $q_0 \in Q$ é o **estado inicial**, por onde o sistema deve começar;
- $F \subseteq Q$ é o conjunto de **estados aceitáveis**.

A Figura 2.2, é um exemplo de uma máquina de estados finita, denominada de M1, que pode ser descrita, segundo a definição formal acima, como M1 = $\{Q, \Sigma, \delta, q_1, F\}$, em que:

- $Q = \{q_1, q_2, q_3\}$;
- $\Sigma = \{0, 1\}$, indicados nas setas de transição na Figura 2.2;
- δ são as transições representadas por setas na Figura 2.2;
- q_1 é o estado inicial, indicado por uma seta que não chega de nenhum outro estado;
- $F = Q$ neste caso, pois pode ser qualquer estado.

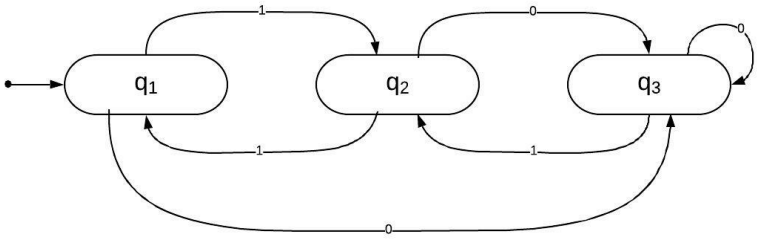

Figura 2.2: Diagrama da máquina de estados M1

Fonte: A autora.

Vale ressaltar que neste trabalho todas as máquinas possuem $F = Q$.

É possível generalizar a definição acima para máquinas de estado finita para incluir máquinas de estados finita probabilísticas (SIPSER, 2013)(LEVELT, 2008). Para isso, no lugar do estado inicial, usa-se uma distribuição de probabilidade $\{p(q_i)\}, \forall q_i \in Q$ em cima do conjunto de estados Q. Além disso, a função de transição de estados δ deve ter como a sua imagem não mais o conjunto Q, mas sim um conjunto o conjunto de todas as distribuições de probabilidade sobre Q. Para ser mais preciso, a função δ é definida de forma que

$$\delta(q_i, \sigma) = \{p(q_j)\}_{\forall q_j \in Q}, \forall q_i \in Q, \forall \sigma \in \Sigma,$$

onde $0 \leq p(q_j) \leq 1$ e $\sum_{\forall q_j \in Q} p(q_j) = 1$.

Este trabalho, contudo, não contempla a implementação de uma máquina de estados probabilística, ficando para trabalhos futuros.

Com o intuito de ampliar o conhecimento sobre máquina de estados, a próxima seção aborda sobre máquina de estados hierárquica, apresentando suas características fundamentais e seus elementos.

2.4 Máquina de estados hierárquica

A origem das máquinas de estados hierárquicas se deu por volta da década de 80, quando Harel (1987) expandiu os diagramas de estado, resultando nos Diagramas de Estado de Harel, que incorporam conceitos de hierarquia entre estados, concorrência

e comunicação.

A incorporação desses conceitos melhorou a estrutura dos Diagramas de Estado e tornou sua linguagem de descrição mais concisa. Essas vantagens proporcionaram aos Diagramas de Estado de Harel a capacidade de serem mais compactos e expressivos, permitindo que diagramas menores representem sistemas com comportamentos mais complexos, de maneira compreensível.

Em seguida, a Object Management Group (OMG) desenvolveu as máquinas de estados UML com o objetivo de superar as principais limitações das máquinas de estado finitas tradicionais, mantendo, no entanto, seus benefícios. Essas máquinas definem um conjunto de conceitos que podem ser usados para modelar comportamento discreto por meio de sistemas de transição de estados finitos (OMG, 2009).

Vale ressaltar que as máquinas de estado UML podem contemplar tanto características de máquinas de Mealy, quanto de Moore. Ou seja, aceitam tanto as ações que dependem do estado e do evento atual, quanto as ações associadas apenas ao estado, que são as ações de entrada e saída.

A OMG determina dois tipos de máquinas de estados, as comportamentais e as de protocolo. A primeira, uma variante baseada em objetos do Diagrama de Estado de Harel, em que foram introduzidos novos conceitos de hierarquia e concorrência, além de expandir a noção de ações, expressa o comportamento de uma parte do sistema. A segunda, por outro lado, expressa o protocolo de uso de uma parte do sistema. Este trabalho se concentra nas máquinas de estados comportamentais, utilizando as noções de hierarquia.

2.4.1 Superestados e subestados

Máquinas de estados tradicionais são ótimas para modelar sistemas simples, porém, se tornam difíceis de manipular à medida que o sistema cresce e se torna complexo (SAMEK, 2008). Uma situação comum é a de que vários estados precisem tratar o mesmo evento da mesma maneira, o que aumentaria significativamente a máquina de estados por ter que incluir as mesmas ações e transições para todos

estes estados. Caso contrário, o estado atual trataria apenas determinados eventos, o que a torna não responsiva a outros. Ou seja, caso ocorra algum evento que o estado não trate, esse será ignorado e erros poderão acontecer.

Para tornar a máquina mais responsiva e simples em sua estrutura, é possível enxugá-la utilizando o conceito de reuso de eventos deste tipo, que se repetem por vários estados. Mas quais estados seriam esses capazes de reutilizar o comportamento? Justamente os estados que caracterizam as máquinas de estados hierárquica: **superestados** e **subestados** (SAMEK, 2008).

O reuso do comportamento de superestados por parte de subestados é uma analogia às classes da programação orientada a objetos. Desse modo, os superestados seriam as classes, e seus subestados herdariam suas funcionalidades, características e operações. O que diferencia o comportamento entre os subestados são suas especializações, porém, estas devem ser coerentes com o superestado no qual estão inseridos a fim de manter a consistência do modelo.

À medida que a complexidade do sistema aumenta, a máquina não aumentará na mesma proporção, uma vez que os subestados poderão reutilizar o comportamento de seus estados superiores. Desse modo, a quantidade de estados e transições diminui consideravelmente, se comparada à quantidade em máquinas de estados tradicionais.

Um dos principais traços da máquina hierárquica é a possibilidade de aninhar estados ilimitadamente, como ilustra o diagrama na Figura 2.3 da máquina M3, em que o estado s_{11} é um subestado do s_1. Neste caso, por conter um subestado, s_1 é chamado de superestado.

Enquanto o sistema está no estado s_{11}, também estará no estado s_1. Dessa forma, se o estado s_{11} não trata um evento, este evento não será descartado como seria em uma máquina tradicional sem a duplicação de transições, e sim enviado para tratamento no estado a nível superior. Neste caso, para o superestado s_1. Esta regra de processamento de eventos pode ser aplicada recursivamente para qualquer nível de aninhamento de estados.

Complementando as nomenclaturas típicas desse modelo, são chamados de **es-**

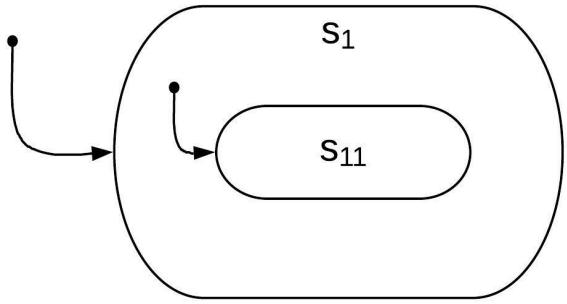

Figura 2.3: Diagrama da máquina de estados hierárquica M3

Fonte: A autora.

tados compostos aqueles que possuem outros estados, e de **estados simples** ou **folhas** aqueles que não possuem. Pode-se acrescentar o **subestado direto**, quando este está aninhado exatamente um nível abaixo, e o **subestado aninhado transitivamente**, quando está aninhado em mais de um nível abaixo (SAMEK, 2008).

Como forma de tornar o entendimento mais fácil, é conveniente supor que todos os estados da máquina estão contidos em um estado composto superior, no maior nível da hierarquia: o estado raiz. Sua indicação nos diagramas é opcional, de modo a não poluir visualmente.

A máquina M4, cujo diagrama encontra-se ilustrado na Figura 2.4, auxilia na exemplificação de como esta configuração pode diminuir a complexidade reduzindo a quantidade de $if - else$ no código. Todos os subestados compartilham do mesmo comportamento que o seu superestado, dessa forma, o que for em comum não precisa ser reescrito, como é o caso da ação que envolve o evento `ev1`. Caso o estado atual seja s_{11} ou s_{12} e o evento a ser tratado seja o `ev1`, estes estados podem ignorá-lo, pois o estado de nível superior s_1 irá tratá-lo.

Segundo Samek (2008), outra leitura que pode ser feita é de que os estados s_{11} e s_{12} reagem igualmente ao evento `ev1`, dessa forma é possível encapsular estes dois estados em um superestado s_1, caracterizando essa propriedade comum entre ambos. Adicionalmente, segundo (HAREL, 1987), o estado s_1 é uma abstração dos estados s_{11} e s_{12}.

Vê-se que modelar um sistema por máquina de estados é uma maneira eficiente de

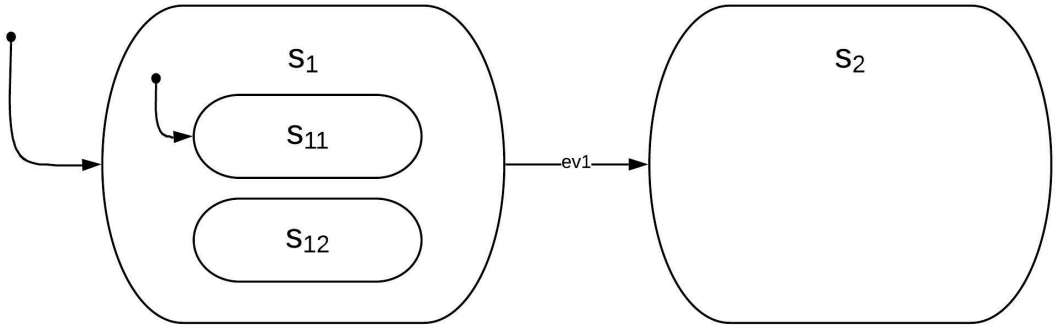

Figura 2.4: Diagrama da máquina de estados hierárquica M4

Fonte: A autora.

descrever seu funcionamento, uma vez que é capaz de restringir seu comportamento aos contextos possíveis, os estados. Estar em um estado significa que o sistema responderá a apenas um subconjunto de eventos permitidos, realizando apenas um subconjunto de transições aceitas, para apenas um outro subconjunto de estados admitidos para o contexto em questão (SAMEK, 2008).

A fim de complementar o entendimento e aprimorar o uso de máquinas de estados hierárquica, a próxima seção discorre sobre outros elementos que a compõe.

2.4.2 Elementos de uma máquina de estados hierárquica

Foi dada anteriormente uma definição formal de uma máquina de estados finita simples. Uma máquina de estados hierárquica herda muito desses conceitos, mas introduz, por sua vez, alguns novos conceitos que são importantes em aplicações práticas. Esta seção apresenta uma descrição mais detalhada dos principais elementos de uma máquina de estados hierárquica, construída em cima do que já foi introduzido na seção anterior. A descrição segue o exposto por Samek (2008).

Estados

Um estado consegue capturar aspectos importantes do sistema, de forma que não é necessário avaliar diversas variáveis para definir em que contexto o programa está, bastando verificar a variável do estado. Da mesma maneira que, para alterar

o contexto, ou seja, o estado, a simplificação é garantida, por não ser preciso modificar várias variáveis. É dessa forma que se reduz a lógica condicional, tornando o programa mais limpo e compreensível (SAMEK, 2008).

O estado é uma forma de condensar toda a história pregressa do sistema em termos dos eventos já ocorridos no passado. Dessa forma, ele representa a memória do sistema. Apesar de não ser possível expressar todo sistema como uma máquina de estados finita, muitas aplicações práticas são compatíveis com este conceito, e nelas o estado atual do sistema resume os efeitos de toda a sequência de eventos ocorridas no passado e determina a reação ao próximo evento a ocorrer.

Eventos

O conceito de evento utilizado para máquinas de estados está associado a ocorrências no tempo e no espaço que são significativas para o sistema. Ocorrências similares, que diferem nos detalhes, podem ser ”agrupadas” como um evento, ou seja, um evento é um tipo de ocorrência. Os detalhes que especificam as ocorrências são seus parâmetros associados, que o identificam quantitativamente (SAMEK, 2008).

O sistema, ao capturar uma instância de evento, a armazena em uma fila de eventos para ser posteriormente processada. Ao despachar a instância para a máquina de estados, esta se torna o evento atual e, após processado e consumido, não se encontrará mais disponível para processamento.

Transições

Outro importante conceito das máquinas de estados hierárquicas são as suas transições. Estas encapsulam as mudanças de estado que ocorrem na máquina em reação à ocorrência de um evento. Elas são uma parte muito importante da máquina porque, como descrito adiante, a elas estão associadas as **ações** que interagem com o mundo externo ou que causam efeitos internos ao sistema.

As transições em uma máquina de estados hierárquica são mais complexas do que

em uma máquina de estados comum, em parte devido à existência de superestados e subestados. Existem quatro tipos diferentes de transição, e seus variados tipos aumentam as possibilidades de tratamento de eventos. A Figura 2.5 mostra exemplos desses tipos de transições:

Transição inicial: indicada por uma seta que não chega de nenhum outro estado. Este tipo de transição só pode acontecer no topo da máquina de estados, onde ela define como a máquina de estados inicia, ou dentro de um superestado, onde ela define para que subestado deve-se ir, se uma outra transição termina no superestado.

Transição local: indicada pela seta que vai de um superestado para um de seus subestados, como a associada ao evento `ev1`. Transições locais são transições que não saem do superestado, e assim, não executam as ações de entrada ou saída do superestado.

Transição interna: indicada pela seta que inicia e termina internamente no mesmo estado, como a associada ao evento `ev2`. Neste tipo de transição, não há mudança de estado, e nenhuma das ações de entrada, saída ou de início são executadas.

Transição externa: também conhecida apenas por transição, indicada pela seta que vai de um estado para um outro, como as associadas aos eventos `ev3` e `ev4`. Este é o caso mais geral de transição, e é o que mais se aproxima a uma transição de uma máquina de estados simples.

Todos os tipos de transição, exceto as transições iniciais, estão, obrigatoriamente, associados a um evento. Uma condição necessária, embora não suficiente, para que uma transição seja realizada é a ocorrência do seu evento.

Vale ressaltar a aplicação das **condições de guarda** para a transição, que é um conceito a ser explicado a seguir. Como lá descrito, quando há uma guarda para determinada transição, esta só será realizada caso a condição seja avaliada como verdadeira. Com o uso de condições de guarda, é possível que um estado possua

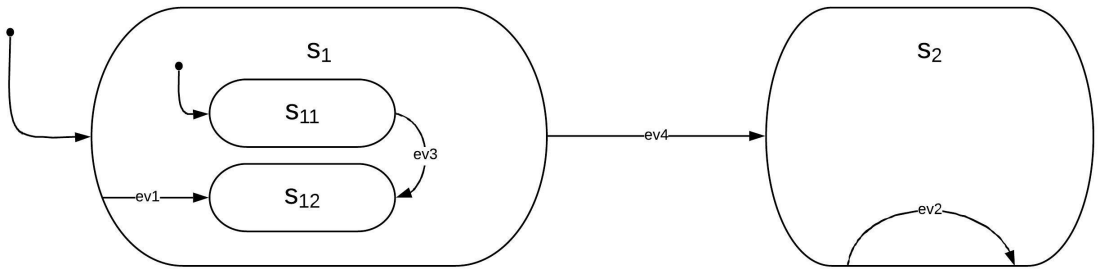

Figura 2.5: Diagrama da máquina de estados hierárquica M5

Fonte: A autora.

mais de uma transição para um mesmo evento, contanto que cada transição tenha condições de guarda distintas. É indicado que a ordem de avaliação dessas condições não seja relevante, e que uma avaliação não interfira em outra, de forma a assegurar a eficiência do programa (SAMEK, 2008).

Condições de guarda

Condições de guarda são testes lógicos que auxiliam nas tomadas de decisão da realização de uma transição. Elas afetam o comportamento da máquina de estados ao permitirem ou não uma transição, ou seja, quando são avaliadas como verdadeiras, a transição ocorre, e quando são falsas, a transição não ocorre (SAMEK, 2008).

A Figura 2.6 apresenta o exemplo da máquina de estados M2, onde a transição $T1$ do estado s_1 para o s_2 só ocorre se a condição de guarda `gc` for verdadeira. A referida Figura mostra a representação gráfica normalmente utilizada para condições de guarda, colocando-a entre colchetes, como uma variável booleana ou uma função de teste que retorna verdadeiro ou falso, `[gc()]`.

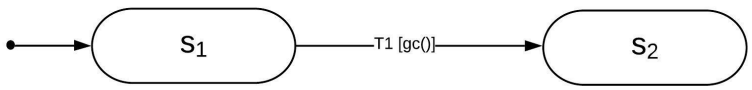

Figura 2.6: Diagrama da máquina de estados M2

Fonte: A autora.

É importante ressaltar que, em princípio, condições de guarda violam a premissa de que o estado de uma máquina de estados encapsula toda a memória do sistema.

Ao permitir que variáveis adicionais possam guardar parte do estado, as condições de guarda introduzem uma camada adicional de complexidade no sistema, que pode ser de difícil visualização em um diagrama gráfico de uma máquina de estados.

Além disso, toda variável, especialmente as booleanas, usadas em condições de guarda para armazenar alguma memória do sistema, pode ser incorporada ao estado, causando, porém, o aumento potencialmente explosivo da quantidade de estados. É importante, portanto, sempre ponderar se condições de guarda serão usadas ou se toda a memória será encapsulada nos estados, pesando as vantagens e desvantagens de cada um.

Usadas judiciosamente, entretanto, condições de guarda podem ser muito úteis na redução significativa da complexidade de uma máquina de estados. Um exemplo clássico é uma máquina de estados que representa uma repetição ou contagem de ações, como, por exemplo, um laço *for* na linguagem C. Para representar esta máquina da forma tradicional apenas com estados, é necessário que a máquina tenha tantos estados quando o número de vezes que o laço executará. Com o uso de uma simples variável inteira, entretanto, é possível representar o laço com um único estado. A redução de complexidade neste exemplo é claramente aparente.

Vê-se então que condições de guarda são uma ferramenta útil na redução de complexidade das máquinas de estado hierárquicas.

Ações

A maneira que a máquina de estados reage aos eventos causando uma mudança na aplicação é chamada de ação. Esta ação pode consistir na alteração do valor de alguma variável, leitura ou escrita nas entradas ou saídas do sistema, realização de uma chamada a uma função ou geração de um outro evento (SAMEK, 2008). Neste último caso, quando um evento é responsável por uma mudança de estado, é realizada uma transição e o evento que ocasionou é chamado de gatilho, ou *trigger* em inglês.

Toda transição pode ter uma ação associada a si que será executada quando a

transição ocorrer, ou seja, quando o evento associado ocorrer e quaisquer condições de guarda associadas forem verdadeiras. Nenhuma ação será executada enquanto um evento não ocorrer.

Cada estado em uma máquina de estados hierárquica pode possuir duas ações especiais que são opcionais: uma ação de entrada e uma ação de saída. Como os nomes dizem, cada ação é executada quando se entra ou sai do estado, quando da mudança de estado causada pela ocorrência de um evento.

Como exemplo, considere a máquina de estados M5 mostrada na Figura 2.5. Se a máquina estiver no estado S_{12} e o evento `ev4` ocorrer, a seguinte lista indica as ações que serão executadas, se presentes, em ordem:

- Ação de saída do estado S_{12};
- Ação de saída do estado S_1;
- Ação da transição de estados;
- Ação de entrada do estado S_2.

Quando da inicialização desta mesma máquina, as ações executadas, se presentes, serão, em ordem:

- Ação da transição inicial da máquina;
- Ação de entrada do estado S_1;
- Ação da transição inicial do estado S_1;
- Ação de entrada do estado S_{11}.

A existência de todas essas ações e a ordem em que são executadas provê flexibilidade para as aplicações de máquinas de estado para se adaptar a diversas situações.

A partir da estrutura gráfica e dos elementos de uma máquina de estados hierárquica, é possível desenvolver um programa computacional que descreva seu comportamento. Posto isto, a próxima seção aborda justamente sobre formas de programação com máquina de estados.

2.4.3 Programação com máquina de estados

Em um programa de controle sequencial, os eventos são aguardados em vários lugares durante sua execução, em que o sistema tanto os pesquisa ativamente, quanto os bloqueia passivamente através de semáforos ou algum outro mecanismo do sistema operacional. Dessa forma, um sistema reativo programado sequencialmente pode ser útil em diversos casos, porém, há outros em que pode apresentar problemas, principalmente quando há diversas fontes de eventos que chegam em tempos e em ordem não determinadas. Enquanto um programa sequencial está esperando que um determinado evento aconteça, ele não está fazendo nenhum outro trabalho e se torna irresponsivo para outros eventos (SAMEK, 2008).

Assim sendo, faz-se necessário uma estrutura de programa que seja capaz de responder a variadas possibilidades de eventos que podem ocorrer em momentos e sequências imprevisíveis. Este tipo de estrutura demanda um pensamento diferente do convencional programa sequencial. Demanda, portanto, uma construção de um programa que não estará no controle enquanto aguarda por um evento, ou seja, não estará ativo enquanto um evento não ocorrer.

Uma vez que um evento ocorra, o programa retoma o controle para tratá-lo e logo retorna ao seu estado de esperar por novos eventos. Os paradigmas de programação orientada a eventos e de programação reativa são especialmente adequados para serem usados aqui. Dessa forma, é possível que o sistema aguarde por vários eventos em paralelo e se mantenha responsivo a todos eles.

Essa estrutura se baseia na divisão do sistema em duas partes, uma para a supervisão dos eventos, que os guarda em uma fila e os despacha, um de cada vez, para a segunda parte, a de tratamento dos eventos (SAMEK, 2008).

Outra importante característica desse tipo de estrutura é a utilização de máquinas de estados para a preservação do contexto do sistema, através do fluxo de eventos e cenários possíveis. A máquina de estados apresenta uma grande vantagem ao adequar o seu comportamento tanto ao evento que chega, quanto ao estado atual, podendo o mesmo evento causar reações diferentes no sistema, de acordo com

o contexto em que o sistema se encontra.

Por sua vez, máquinas de estados necessitam de uma infraestrutura mínima para operar confiavelmente. Infraestrutura esta com, por exemplo, enfileiramento e temporizações baseados em eventos. É importante observar também que, enquanto um evento está sendo tratado, a máquina de estados está em um estado inconsistente, e portanto incapaz de fazer qualquer outra coisa. Em outras palavras, a parte do código que trata um evento forma uma seção crítica, ou região crítica, um conceito importante de computação paralela (HERLIHY; SHAVIT, 2008). Assim, deve-se tomar cuidado ao programar uma máquina de estados devido à ocorrência assíncrona de eventos.

Uma maneira de controlar o fluxo de um programa é com a utilização de vários *if-elses* aninhados, variáveis globais e *flags*, na intenção de garantir o contexto adequado para o tratamento de determinado evento. Porém, este método torna o código complexo de avaliar e desnecessariamente longo em tempo de execução. Acrescenta-se também o fato de tornar o programa mais suscetível a erros com o aumento da complexidade.

Há a possibilidade da eliminação de um número significativo de condicionais, trazendo, dessa forma, o benefício de tornar o programa mais inteligível, mais fácil de testar, de realizar manutenções e da redução da quantidade de caminhos possíveis de execução. Vale ressaltar que ao considerar sempre, não somente o evento, mas também o contexto atual, o sistema ganha ainda mais em confiabilidade (SAMEK, 2008).

Nessa abordagem, ao utilizar uma DSL, é possível gerenciar tal complexidade na descrição de uma máquina de estados hierárquica. Ao ser escolhido como um dos métodos utilizados neste projeto, esse assunto é o tema da próxima seção.

2.5 Linguagem de domínio específico

Linguagem é definida pelo dicionário Aurélio como a expressão do pensamento pela palavra ou por outros meios (FERREIRA, 2004). Uma linguagem é o que

permite os seres humanos se comunicarem de uma forma rica e eficiente, garantindo a transmissão de ideias elaboradas e permitindo a realização de tarefas complexas.

Devido à sua utilidade, linguagens também são usadas na programação de CPU. Estes sistemas digitais têm um conjunto próprio de ações que são capazes de realizar. Para que seja possível especificar estas ações, contudo, a cada ação é associada uma palavra-binária, denominada de instrução, para representá-la. Uma sequência destas instruções é denominada de um programa binário, ou simplesmente programa para a CPU, que define a sequência de ações a serem por ela executadas (AHO et al., 2007).

Embora estas palavras-código sejam prontamente inteligíveis para a CPU, desde o início do uso das CPUs ficou claro que, para um ser humano, é um desafio ter que lidar com elas. Um ser humano reconhece com mais facilidade sequências de palavras do que sequências de números, e portanto logo foram desenvolvidas linguagens de programação que representassem as instruções por palavras para permitir uma melhor expressão das ideias de um programa pelas pessoas. Uma sequência destas palavras também é denominada de um programa para a CPU, mas também recebe o nome, para diferenciar da sequência de instruções binárias, de código-fonte do programa. Enquanto que a sequência de instruções binárias é chamada de código-binário do programa.

Inicialmente, estas linguagens eram um mero auxílio mnemônico para as instruções. Esse tipo de linguagem é chamado de Assembly, e geralmente tem uma palavra código para cada instrução da CPU (WIKIPEDIA, 2024). A conversão das palavras para as instruções binárias é feita através de uma relação uma para uma com o auxílio de um programa chamado de Assembler.

Com o passar do tempo, no entanto, foi ficando claro que as linguagens poderiam ser usadas para expressar conceitos lógicos mais abstratos e poderosos do que as simples operações lógicas e aritméticas básicas definidas pela CPU. Assim, foram criadas linguagens ao longo do tempo cada vez mais versáteis e capazes de expressar de forma clara conceitos lógicos e abstratos, progressivamente mais poderosos.

Essas linguagens, entretanto, apesar de terem evoluído e se tornado bastante diferentes das linguagens primitivas iniciais, ainda podem codificar qualquer ideia que era antes escrita através das instruções binárias da CPU (embora, talvez, nem sempre da forma mais eficiente). Por essa razão, essas linguagens são denominadas de linguagens de uso geral, e são universalmente usadas hoje em dia (FOWLER, 2010).

Todas elas são, todavia, voltadas para o entendimento do ser humano, necessitando que sejam convertidas para as instruções binárias de uma CPU através de um programa especial chamado de compilador ou, em alguns casos, um interpretador, quando se converte à medida que se executa o programa, muitas vezes em uma CPU virtual.

Devido às limitações destes programas, porém, as linguagens precisam ter uma sintaxe bem definida para não haver ambiguidades e permitir ao compilador realizar a conversão de forma precisa e eficiente. A língua nativa dos seres humanos, como o português, o francês ou o inglês, possuem ambiguidades demais para serem úteis nesta tarefa. Não obstante essa desvantagem, as linguagens de programação atuais conseguem ficar próximas o suficiente das linguagens naturais humanas para, ao mesmo tempo, permitir o uso confortável pelos humanos e uma fácil interpretação pelo programa compilador, o que as torna extremamente úteis na sociedade atual.

Infelizmente, apesar deste crescente avanço das linguagens de uso geral, elas ainda são complexas o suficiente para que o seu aprendizado não seja trivial. Além disso, devido à sua flexibilidade, erros lógicos podem se infiltrar no código-fonte, que são muitas vezes difíceis de serem localizados. Assim, em várias situações, é desejável abrir mão da flexibilidade de uma linguagem de uso geral em prol de uma maior simplicidade e facilidade de aprendizado, mantendo ainda assim a vantagem de ser útil para o usuário. Uma DSL é uma linguagem que segue essa orientação (FOWLER, 2010), e consiste, portanto, de uma gramática que restringe as expressões, ou programas, que podem ser escritas a partir dela para simplificar esse processo de escrita. A gramática, entretanto, deve ainda ser rica o suficiente para expressar os

conceitos na área do domínio em questão.

As DSL são projetadas para permitirem que seus usuários sejam mais eficientes em um certo domínio de operação. Exemplos de uso de DSL podem ser encontrados em um amplo espectro de aplicações, algumas das quais são, até mesmo, inesperadas. Inúmeras DSL já foram criadas para definir formatos de dados, formatos de arquivos de configuração, protocolos de rede, padrões de proteínas, sequências genômicas, linguagens de controle de sondas espaciais, entre outras aplicações (FOWLER, 2010).

DSL são particularmente importantes no desenvolvimento de *software*, pois representam um meio de codificar um problema de um modo mais natural para o usuário, além de robusto e fácil de manter, em comparação à escrita do mesmo projeto em uma linguagem de uso geral. Como a DSL só precisa ser capaz de expressar problemas em um domínio de operação, é possível trocar a flexibilidade e a generalidade de uma linguagem como C pela simplicidade da DSL.

A título de exemplo, há a linguagem de programação P, desenvolvida pela Microsoft Research, com o objetivo de facilitar o desenvolvimento de sistemas distribuídos e concorrentes de forma segura e confiável. Esta linguagem combina recursos de modelagem formal e programação convencional, fornecendo suporte nativo para especificação de protocolos, verificação de propriedades e geração automática de código. A linguagem P oferece abstrações poderosas para lidar com eventos, estados e comunicação assíncrona, permitindo aos desenvolvedores expressar de forma concisa e precisa o comportamento de sistemas complexos. Com sua ênfase na segurança e confiabilidade, a linguagem P é uma ferramenta valiosa para o desenvolvimento de sistemas críticos e escaláveis (GUPTA et al., 2012).

A Microsoft utilizou a linguagem P para implementar e verificar o *drive* de dispositivo USB do Windows 8, publicando em (GUPTA et al., 2012) que o *driver* resultante apresenta maior confiabilidade e melhor desempenho do que sua versão anterior, que não utilizava P.

Um outro exemplo vem da Agência Espacial americana — NASA (do termo em inglês *National Aeronautics and Space Administration*), que usa uma DSL de co-

mando para missões espaciais para melhorar a confiabilidade, reduzir o risco e o custo, e aumentar a velocidade de desenvolvimento (KUNG, 2022). Além disso, a NASA desenvolveu a DSL Proteus (NASA, 2020)(TELLIER et al., 2020) especificamente para implementar máquinas de estado hierárquicas para uso na escrita de software de controle e simulação, incluindo o do seu veículo explorador *Curiosity*. O código do Proteus, entretanto, não parece estar disponível, e as informações sobre seu projeto não são detalhadas.

Outras DSLs também foram especificamente projetadas para representar máquinas de estado. A maioria, entretanto, não se aplica a máquinas de estado hierárquicas. A linguagem Ragel (THURSTON, 2024) é uma linguagem para especificar máquinas de estado em função de Expressões Regulares (FRIEDL, 2006). Por esta razão, sua gramática não é tão limpa como outros casos. Para alguém que não é acostumado com expressões regulares, pode ser difícil programar uma máquina de estados. Outra desvantagem é que o compilador para esta linguagem depende da linguagem Colm (THURSTON, 2021), que não é muito conhecida. Assim, qualquer modificação no compilador é de difícil implementação por um programador que não conhece a linguagem Colm.

Uma outra linguagem voltada para máquinas de estados é o desenvolvido pelo State Machine Compiler (RAPP, 2021). Este é um programa que converte a especificação de uma máquina de estados escrita em sua DSL em código para uma variedade de linguagens. Apesar de robusto e gerar código para muitas linguagens, a linguagem não é voltada para máquinas hierárquicas.

Para referência, uma outra linguagem definida para máquinas de estados hierárquicas é a SCXML (W3C, 2015), do termo em inglês *State Chart XML*. Como o nome indica, esta linguagem não é uma DSL, e sim uma linguagem baseada em XML. É, porém, uma iniciativa importante nessa área, e portanto merece a menção neste trabalho. Apesar da importância dessa linguagem, ela sofre das desvantagens da linguagem XML, e é uma linguagem mais voltada para processamento pelo computador do que para leitura pelos seres humanos, além de ser um pouco prolixa e

gerar arquivos maiores do que o necessário. Uma DSL apropriada pode evitar esses dois problemas. Por estas razões, esta linguagem não será mencionada no erstante do trabalho.

Uma importante observação é que o compilador de uma DSL, não precisa, necessariamente, converter o código-fonte para um código-binário. De fato, muitas DSL têm seus programas convertidos para o código-fonte de uma outra linguagem, possivelmente de uso geral. Isto simplifica bastante a construção do compilador, uma grande vantagem na prática e que será usada plenamente neste trabalho.

Portanto, neste trabalho, uma DSL será definida e usada para fornecer uma maneira fácil e eficiente de especificação de uma máquina de estados hierárquica. Será também criado um compilador para converter esta especificação para um programa na linguagem C que implemente esta máquina de estados.

Com o propósito de definir uma DSL, é necessário apresentar dois conceitos importantes para sua construção. As próximas sessões são dedicadas a abordá-los, a gramática, responsável por definir a sintaxe da DSL delineada, e o analisador sintático, responsável por interpretar a gramática.

2.5.1 Gramática

A definição de qualquer linguagem se dá através de sua gramática. Formalmente, a gramática é descrita como um objeto matemático que permite especificar uma linguagem, ou seja, é um conjunto finito de regras de formação de sentenças em uma linguagem (FOWLER, 2010). Estas regras definem a sintaxe da linguagem, ou seja, elas guiam como transformar um texto de entrada em uma *árvore de sintaxe*, conceito a ser elucidado na próxima seção. Ademais, as regras são compostas por um termo e uma declaração que as determinam. Dessa forma, todos os termos aceitos compõem a gramática da linguagem. Vale ressaltar, ainda, que todas as palavras aceitas pela linguagem formam o alfabeto, ou vocabulário, da gramática.

É importante observar que uma gramática é uma referência para pessoas que desejam aprender a sintaxe da linguagem, e até mesmo a gerar o seu compilador,

pois materializa um completo entendimento do procedimento de construção das palavras. A maioria dos compiladores e interpretadores precisam extrair o significado do programa, para assim gerar o código compilado ou realizar a execução interpretada.

Como exemplo, a declaração de uma adição pode utilizar a regra `adicao = numero '+' numero` para compor determinada gramática. Destaca-se a possibilidade de regras mencionarem outras regras, ou seja, é possível que seja criada uma regra para determinar quais caracteres podem ser considerados `numero`, e assim validados, para serem utilizados (FOWLER, 2010).

Vale ressaltar que a gramática apenas define a forma correta como uma sentença deve ser escrita, mas não diz nada sobre o seu significado. Para exemplificar, é possível considerar a seguinte declaração: `1 + 2`. A depender do contexto, pode significar `3`, caso a operação seja interpretada como adição de dois números, ou `12`, caso a operação seja interpretada como concatenação de duas *strings*. Ambas as interpretações são válidas, e vai depender da aplicação decidir qual deve usar. Note que, em ambos os casos, a gramática é a mesma, porém com semânticas distintas.

Assim, é possível afirmar que a gramática apenas define a sintaxe da linguagem, contudo, não possui nenhuma informação sobre sua semântica. A sintaxe tem a função de capturar as expressões válidas para o programa. Nesse sentido, a gramática não tem a intenção de descrever o significado das sentenças, apenas de definir sua possíveis formas.

O processo completo da interpretação de uma DSL pode conceitualmente ser dividido nas fases de análise léxica, onde se converte o texto original em uma sequência de *tokens*, análise sintática, que fornece a árvore de sintaxe abstrata, e a análise semântica, que extrai o significado do programa (PARR, 2010). Dentro deste contexto, a gramática e o analisador sintático (ou *parser*, em inglês) estão intimamente conectados. Com o intuito de explanar melhor sobre essa relação, a próxima seção tem por objeto elucidar o processo de análise sintática, processo este que deve ter como base uma especificação precisa da linguagem, a sua gramática.

2.5.2 Analisador sintático

Análise sintática, ou *Parsing*, em inglês, é o processo de reconhecimento da estrutura de uma linguagem. Apresenta grande importância para os compiladores, pois sua função é descobrir o que a entrada significa, identificando cada parte do programa escrito (PARR, 2010).

As análises lexical, sintática e semântica fazem parte desse processo de compilação, sendo a primeira a investigação da entrada de acordo com o alfabeto definido pela gramática, a segunda onde se analisa a entrada para verificar sua estrutura gramatical, e a última a busca por erros semânticos.

O primeiro passo, a análise lexical, é também chamado de geração de *tokens*. Neste passo o programa escrito é examinado e dividido dentre as estruturas definidas pela gramática. Em seguida, é realizada a análise sintática, na qual é verificado se os *tokens* formam uma expressão válida, averiguando inclusive a ordem em que devem ocorrer. Por último, a análise semântica, nesta etapa, ocorre a elaboração das implicações das expressões validadas e toma-se a ação apropriada.

Fundamentado na gramática, o analisador sintático poderá gerar uma estrutura de dados com base no texto de entrada. Esta estrutura, chamada de árvore sintática, é de grande importância para o processamento do texto, pois a partir dela é possível identificar os blocos definidos pela gramática formal.

As árvores, também conhecidas como AST (*Abstract Syntax Trees*), são representações altamente codificadas da entrada, pois codificam informações essenciais, como as regras definidas pela gramática. Ao trabalhar com essa versão, o processo torna-se muito mais fácil de lidar.

A formação dessa árvore se dá de maneira simples. As regras definidas pela gramática são representadas pelos **nós internos** da árvore. Seus filhos, os galhos que partem do nó, podem ser ou outras regras ou **terminais**. Terminais são definidos como uma sequência de ***tokens***, ou seja, os símbolos válidos do vocabulário. Estes, são os **nós folha** que, por definição, não possuem filho. Ou seja, a análise sintática utiliza a gramática formal para transformar uma sequência plana de símbolos em

uma árvore bidimensional (PARR, 2010).

Com o intuito de facilitar a visualização deste conteúdo, deve-se ter em mente que a linguagem é apenas uma série de sentenças válidas. Dessa forma, para verificar se a sentença faz parte da linguagem, é necessário que o analisador sintático identifique a árvore desta sentença. Uma breve adaptação do exemplo de (PARR, 2010) pode ser apresentado a seguir para visualização de uma AST. Uma simples entrada, como **`return x + 1;`** de acordo com a gramática descrita na Figura 2.7, deverá gerar a árvore apresentada na Figura 2.8.

```
1  state        :    returnstate          ;
2  returnstate  :   'return' expr ';'   ;
3  expr         :   'x' '+' '1'          ;
```

Figura 2.7: Exemplo de gramática

Fonte: A autora.

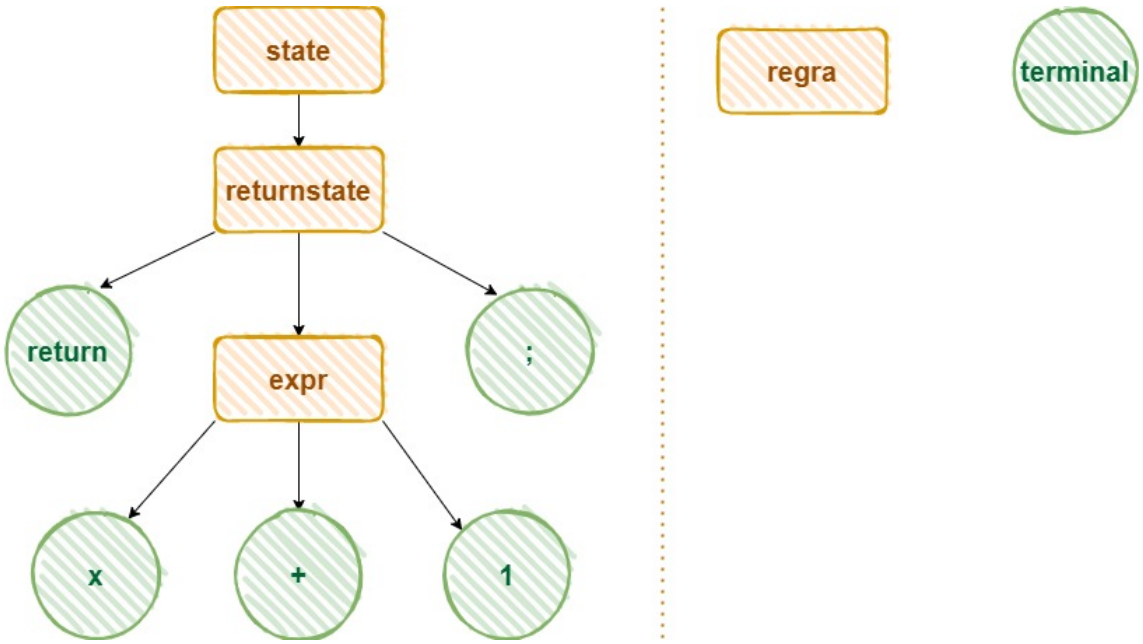

Figura 2.8: Exemplo de estrutura em árvore

Fonte: A autora.

Vale evidenciar a possibilidade de manipulação da árvore sintática de várias maneiras, à medida que se caminha por seus "galhos". Dessa forma, a AST revela-se como uma melhor representação do texto de entrada do que uma representação por

palavras. Um artifício que muito contribui para o processo de reconhecimento da linguagem são os geradores de analisadores sintáticos, que o geram a partir de uma tradução da gramática (PARR, 2010). Neste trabalho, o Lark foi utilizado como gerador da AST (LARK, 2022), ferramenta elucidada no capítulo subsequente.

Capítulo 3

Métodos e Materiais

Este capítulo descreve a metodologia e os materiais e técnicas utilizadas neste trabalho. A metodologia é descrita primeiro, e envolve alguns passos tradicionais do processo de desenvolvimento de um produto de engenharia. Em seguida, é descrito o que foi usado para implementar o compilador que é o foco do trabalho.

3.1 Metodologia

A metodologia escolhida para este trabalho pode ser separada em 3 fases: projeto, desenvolvimento e validação. Estas fases foram necessárias para se obter um produto funcional e robusto, e são baseadas nas etapas comuns a um processo tradicional de desenvolvimento de um produto.

3.1.1 Projeto

Na fase de projeto, foi estabelecida qual seria a estrutura da linguagem a ser desenvolvida. Como será detalhado na seção 4.1, foi determinado quais os elementos a serem considerados na linguagem, a saber: estados, eventos, transições, ações e condições de guarda.

Em seguida, foi definido como identificar cada um desses elementos e como o usuário deveria organizá-los para descrever sua máquina de estados. Com esse modelo estabelecido, foi possível determinar uma gramática completa.

3.1.2 Desenvolvimento

A partir do projeto estruturado, foi possível iniciar a fase de desenvolvimento. Inicialmente, foi desenvolvida a gramática de acordo com a seção 2.5.1, definindo as regras e terminais de acordo com a biblioteca Lark, criando, dessa forma, a nova linguagem a ser utilizada para descrever máquinas de estado hierárquicas.

Em seguida, foi realizada a implementação do programa responsável por ler a máquina de estados descrita na DSL criada e gerar a estrutura em linguagem C, para uso do usuário. O programa foi dividido em duas partes: a primeira lê a árvore e separa suas partes, como estados, eventos e transições, especificada na seção 4.2; e a segunda, que utiliza essas partes para gerar o código final, apresentada a partir da seção 4.3.

Paralelamente ao desenvolvimento do programa, foram realizados testes para validar cada funcionalidade, evitando a descoberta de erros apenas no final do processo. Dessa forma, foi garantido o correto funcionamento de todas as partes do programa.

3.1.3 Validação

Por fim, foi realizada a etapa de validação, para verificar se o projeto alcançou seus objetivos. Após a integração das duas partes mencionadas anteriormente, foram utilizados exemplos de máquinas de estados escritas na linguagem criada e foi verificado se o programa gerado, associado a essa máquina de estados, estava correto.

Para ilustrar de forma prática, foi selecionado um projeto em que o funcionamento foi modelado por uma máquina de estados e seu *firmware* foi embarcado em uma placa desenvolvida especificamente para este trabalho. O exemplo completo está disponível no capítulo 5.

3.2 Materiais

As ferramentas utilizadas para a construção deste trabalho foram a linguagem de programação Python e a biblioteca de *parsing* Lark, disponível para Python, que viabiliza a construção de uma gramática e geração automática de sua árvore de sintaxe. A seguir, estas duas ferramentas serão descritas.

3.2.1 Python

Python é uma linguagem de programação que teve seu projeto iniciado na década de 1990 por Guido van Rossum, um matemático e programador holandês. Como objetivos para esta linguagem, Guido van Rossum definiu que esta fosse de fácil compreensão e uso, inteligível como o inglês, intuitiva e tão eficiente quanto outras de alto nível. Sua sintaxe permite que um código seja escrito em menos linhas do que seu equivalente em C, por exemplo (NOSRATI, 2011).

Atualmente é mantida pela *Python Software Foundation* e, embora a empresa seja responsável pela linguagem, o desenvolvimento do Python também sofre influências dos usuários, uma vez que a linguagem é de código aberto, permitindo que pessoas da comunidade contribuam para seu melhoramento (PYTHON, 2022).

Vale evidenciar que Python é uma linguagem de uso geral, o que contribui para sua popularidade entre programadores. Alguns dos aspectos que justificam seu uso extensivo é o suporte a diferentes paradigmas de programação, como programação orientada a objetos ou programação estruturada (NOSRATI, 2011).

Mais um importante aspecto que aumenta a sua popularidade e acelera a criação de programas em Python se deve à sua modularidade, sua vasta variedade de bibliotecas, que são compatíveis com ferramentas de outras linguagens, acarretando na simplificação da integração com outros sistemas, melhorando sua compatibilidade (LINGE, 2019).

Destaca-se a importante característica do Python de ser um sistema com tipagem dinâmica e gerenciamento automático de memória (NOSRATI, 2011), tirando do programador a preocupação com a alocação de espaço de memória das variáveis.

Os recursos da linguagem abstraem esse trabalho e, embora apresente um custo em sua performance, há o benefício para o programador, que poderá focar apenas no algoritmo a ser implementado.

O modo como a linguagem foi desenvolvida permite, então, que em curto espaço de tempo, diversos tipos de programas, simples ou com complexidade superior, sejam implementados, até mesmo por programadores iniciantes. Há diversas bibliotecas importantes e estáveis, que facilitam o uso da linguagem em áreas distintas, como: matemática, aprendizagem de máquina, análise e manipulação de dados, biologia, desenvolvimento *web*, visão computacional, apenas para citar alguns (GORELICK, 2014).

O Python fornece uma variedade de estruturas de dados padrão de grande utilidade, das quais, as mais utilizadas neste projeto foram as listas, tuplas e dicionários. Outros artifícios empregues foram os geradores e iteradores. As estruturas e artifícios citados têm seus aprofundamentos nas seções a seguir.

Listas e tuplas

Listas e tuplas são casos especiais de sequências em Python, compondo uma classe de estrutura de dados de grande importância para a linguagem. Dado sua importância, o Python acaba por generalizar diversas das propriedades desses dois casos especiais em estruturas denominadas de iteradores e geradores, propriedades abordadas adiante. Ademais, uma sequência é simplesmente uma lista de dados com alguma ordem intrínseca. A ordem é essencial pois, sabendo a posição, é possível recuperar o dado. Pode-se acrescentar que a diferença entre as duas formas de sequências citadas é a seguinte: as listas são dinâmicas, enquanto que as tuplas são sequências estáticas (GORELICK, 2014).

As listas são sequências de objetos arbitrários, não necessariamente do mesmo tipo, dispostos em ordem numérica sequencialmente a partir do zero. Por ser uma estrutura de dados dinâmica, a lista tem a característica de ser mutável. Ou seja, é possível acessar, alterar, acrescentar ou remover objetos em qualquer posição (LINGE,

2019).

As tuplas apresentam semelhanças com as listas, como citado anteriormente, pois são coleções de objetos. Entretanto, sua diferença consiste em ser imutável, ou seja, não poderá ser modificada. A tupla pode ser vista como um objeto único, formado por partes diversas, que não poderá ser removido, alterado ou acrescido de novas partes. Diante disto, uma tupla ocupa menos espaço na memória do que uma lista.

Dicionários

Dicionários, em Python, se apresentam como uma excelente maneira de armazenar e trabalhar com dados em um programa. Seu uso é ideal quando os dados não possuem uma ordem intrínseca, mas possuem um único objeto, que pode ser uma referência para este dado. Esta referência é também chamada de chave, e o dado de valor do dicionário (LANGTANGEN, 2016).

Logo, este tipo de estrutura, que também se comporta como uma sequência, é um bom caminho para armazenar dados, que podem ser indexados por uma chave. E, por ser dinâmico, é válido a inserção, remoção e modificação de seu conteúdo. É de referir que as chaves de um dicionário, devem ser elementos imutáveis, o que também justifica a criação das tuplas, em adição às listas, na linguagem Python.

Iteradores e geradores

A capacidade de iteração sobre um objeto é um dos artifícios mais importantes em Python. Ademais, a forma mais comum de iteração é um simples *loop* sobre todos os membros de uma sequência, como uma *string*, uma lista ou tupla (LANGTANGEN, 2016).

A declaração de *looping* mais utilizada é o *for*, empregue para iterar sobre uma coleção de itens. Como resultado, a cada iteração do *for*, o valor declarado assume, sucessivamente, um item da sequência, até que todos os itens sejam varridos.

Igualmente significativos, os geradores são bastante úteis quando a intenção é que uma função retorne não apenas um, mas sim uma sequência de objetos, fruto

de suas iterações internas. Tal façanha em Python é alcançada ao utilizar o artifício *yield*. Por consequência, qualquer função que usufrua desse artifício, será chamada de geradora (BEAZLEY, 2001).

No desenvolvimento de programas que processam *pipelines*, *streams* ou *dataflow*, os geradores se apresentam como um poderoso recurso. São normalmente utilizados com outros objetos iteráveis, como listas e arquivos, tal qual o presente projeto. À vista disso, o gerador é capaz de retornar todos os itens de uma lista, ou as linhas de um arquivo, após o tratamento dentro da função geradora.

Ao utilizar geradores e iteradores em conjunto, é possível processar muito mais dados utilizando pouco espaço de memória. Embora os resultados sejam avaliados lentamente, apenas os dados necessários são processados, não necessariamente armazenados, a não ser que requeridos explicitamente (GORELICK, 2014).

Além da economia de memória, pode-se citar o benefício da maior agilidade de processamento, quando comparado ao manuseio com listas, uma vez que não haveria a necessidade de acrescentar objetos a esta estrutura, economizando assim, o uso do recurso de adição de objetos. Do mesmo modo, o programa torna-se mais simples de ler e entender, refletindo na qualidade do código.

A seção seguinte aborda a biblioteca em Python aplicada para a implementação do compilador desenvolvido no projeto. Ressalta-se, mais uma vez, a riqueza de recursos que a linguagem oferece para os mais variados intentos.

3.2.2 Lark

Como instrumento de geração do compilador deste trabalho, foi utilizada a biblioteca Lark, desenvolvida para a linguagem de programação Python (LARK, 2022). A utilização do Lark descomplica o desenvolvimento do compilador, ao tornar a implementação do analisador de sintaxe um processo simples e o mais genérico e flexível possível.

O Lark foi implementado de forma que a gramática é escrita separadamente do código, no padrão de definição de regras e terminais, já mostrada nesse trabalho,

facilitando assim a escrita e leitura da gramática. A saber, esse padrão chama-se EBNF (Extended Backus–Naur form).

Outras duas facilidades do Lark são a geração da árvore sintática, devido a todas as facilidades que se tem ao trabalhar com elas, assim como, a aceitação de qualquer gramática livre de contexto, ou seja, basicamente qualquer gramática que se possa escrever no padrão EBNF. Dessa forma, facilita para os iniciantes, mas também contempla os que precisam de uma ferramenta mais completa.

Sua documentação estabelece uma gramática como uma lista de regras e terminais. Por sua vez, regras e terminais são definidos da seguinte maneira:

Regras: definem a estrutura da árvore sintática; pode ser uma lista de outras regras, terminais, literais, expressões regulares e os operadores gramaticais;

Terminais: definem o alfabeto da linguagem; podem ser compostos por literais e/ou uma concatenação destes com outros terminais, porém não é permitido regras; são usados para fazer a correspondência entre o texto de entrada, com os símbolos do alfabeto da linguagem e outros terminais; a recursividade não é permitida.

Como literais entende-se as *strings*, as expressões regulares e operadores gramaticais, tais como $+$, ?, $*$ e $|$. Ademais, intervalos de literais também são aceitos, como "a"..."z" ou "1"..."9".

A árvore que o Lark automaticamente constrói, como já dito anteriormente, tem como base a estrutura definida pela gramática. Em vista disso, cada regra encontrada no texto vem a ser um **nó** ou **ramo**, na árvore (no capítulo 2.5.2 chamou-se de nó interno); e seus filhos, terminais ou regras, ramificam a partir deste nó. Caso um dos filhos seja um terminal, este será representado por uma **folha**, na árvore.

A declaração de um terminal ou de uma regra em uma gramática Lark, se dá a partir de uma lista de definições e diretrizes, cada uma em sua própria linha. A nomeação segue o seguinte padrão: uma regra deve conter apenas letras minúsculas e um terminal apenas letras maiúsculas. Essa distinção tem efeito prático na geração da estrutura da árvore resultante (LARK, 2022).

Com o intuito de aumentar a usabilidade e clareza do código, há tipos pré-definidos na biblioteca que definem regras gramaticais para os terminais. Neste trabalho, foram utilizados dois tipos, sendo eles:

CNAME: (*Contextual Name*) é uma construção específica usada para atribuir um nome a um padrão análogo à variáveis em programação, pois aceitam caracteres alfanuméricos sem espaço entre eles;

ESCAPED_STRING: criada para representar uma sequência de caracteres delimitada por aspas, analogamente às *strings* em linguagem de programação. Essa notação permite representar *strings* que contenham caracteres especiais.

A Figura 3.1 apresenta um breve exemplo da sintaxe aceita para definição de uma gramática.

```
regra:  <EXPRESSÃO>
        | etc           // A definição pode ser estendida
        | etc           // para a próxima linha utilizando
        | etc           //  o operador ou: |

TERMINAL: <EXPRESSÃO>   //Regras não são permitidas
```

Figura 3.1: Exemplo de gramática genérica

Fonte: A autora.

A Figura 3.2, ilustra a definição de uma gramática, para reconhecer uma declaração específica de *(if .. then ..)*, em que é possível visualizar a declaração de regras e terminais, usando também tipos pré-definidos como `WORD` e `NUMBER`, que representam palavras e números, respectivamente.

```
from lark import Lark, tree

grammar = '''start: ifstatement
            ifstatement: "if" exp "then" assign ";"

            exp: WORD OPERATOR NUMBER
                | WORD OPERATOR WORD

            assign: WORD "=" NUMBER
                    | WORD "=" WORD

            OPERATOR: (">"|"<")


            %import common.WORD   // imports from terminal library
            %import common.NUMBER
            %ignore " "           // Disregard spaces in text
         '''

parser = Lark(grammar)

sentence = "if x<0 then x=0;"

print(parser.parse(sentence).pretty())
```

Figura 3.2: Gramática (if .. then ..)

Fonte: A autora.

Como a árvore gerada pelo analisador de sintaxe mostra as regras e os terminais, espera-se que ao final do programa seja impressa na tela a estrutura da árvore com estes ramos e folhas da frase `"if x<0 then x=0;"`. Desse modo, a resposta que o programa oferece está representada na Figura 3.3.

```
start
  ifstatement
    exp
      x
      <
      0
    assign
      x
      0

>>>
```

Figura 3.3: Árvore gerada pelo analisador de sintaxe

Fonte: A autora.

Este é o modo de visualização mais agradável. Mas, também pode-se visualizar o vetor que dá origem a essa árvore, e é desse vetor que pode-se caminhar por seus galhos e extrair qualquer informação que se queira. Dessa forma, ao enviar apenas o comando `print(parser.parse(sentence))`, obtém-se o vetor ilustrado na Figura 3.4.

```
Tree(Token('RULE', 'start'), [Tree(Token('RULE', 'ifstatement'),
[Tree(Token('RULE', 'exp'), [Token('WORD', 'x'), Token('OPERATOR', '<'),
Token('NUMBER', '0')]), Tree(Token('RULE', 'assign'), [Token('WORD', 'x'),
Token('NUMBER', '0')])])])
>>>
```

Figura 3.4: Vetor gerado pelo parser

Fonte: A autora.

Nota-se que o Lark separa a regra em dois campos, o primeiro como um *token*, separando o nome da regra em questão, `Token("Rule", "nome_da_regra")`, e o campo seguinte como uma lista com todos seus filhos, representada por uma *Tree*, da seguinte maneira, `[Tree(Token("RULE ou TERMINAL"), "nome")]`. Caso este último componente seja também uma regra, ele também terá dois campos, um como *token* para seu nome, e o seguinte para seus filhos, e assim sucessivamente.

Explanadas então as ferramentas utilizadas no projeto, é possível, desse modo,

abordar a implementação deste ferramental no presente trabalho, como apresenta o próximo capítulo.

Capítulo 4

Implementação

Como descrito no Capítulo 3, o programa consiste basicamente de um analisador de sintaxe (um *parser*, em inglês), com uma gramática apropriada, seguido de um analisador de semântica e um gerador de código, para gerar o código final em C, implementando a máquina de estados. Assim, neste capítulo será apresentado como o programa deste trabalho foi tecido. Inicialmente, será retratado o desenvolvimento da gramática para a nova linguagem. Na sequência, será exposta a forma como o programa, a partir de um texto que descreve uma máquina de estados, de acordo com a gramática desenvolvida, gera o código de descrição da máquina de estados e, por último, a maneira como o programa foi estruturado para atender às necessidades do tema.

4.1 Gramática

A maneira mais simples de representar uma máquina de estados hierárquica é a partir de seu diagrama gráfico, como exemplificado na Figura 2.4 do Capítulo 2. Nesse diagrama é possível visualizar todos os estados e transições. Porém, uma descrição textual de fácil entendimento também é possível de se alcançar. Um exemplo está representado na Figura 4.1.

```
[*] -> s1 :          # Transição inicial da máquina de estados

s1 {
    s11 {
    }
    s12 {
    }
    [*] -> s11 :     # Transição inicial para o estado s11
    s1 -> s2 : ev1   # Ou seja, transição do estado s1 para s2
                     # caso ocorra o evento ev1
}

s2 {
}
```

Figura 4.1: Exemplo de descrição de máquina de estados hierárquica

Fonte: A autora.

Esse exemplo define uma máquina de estados com quatro estados, s_1, s_2, s_{11} e s_{12}, onde s_1 é um super-estado. Em relação à transição, dentro da descrição do estado s_1 há o evento `ev1` que realiza a transição para o estado s_2, indicado pela seta representada por ->. Há também a transição inicial da máquina de estados e a transição inicial do super-estado s_1.

Para a criação da gramática desta linguagem, é necessário identificar os elementos que são estados ou transições. Dessa forma, pode-se acrescentar tanto o tipo de variável “state” quanto o tipo “transition”, que na gramática serão definidas como regras. Além disso, pode-se definir também a transição inicial de um super-estado, ou estado-raiz, representada por [*], que na gramática será definida como um terminal.

Como descrito na Seção 2.4.2, as transições têm associadas a si o evento que as disparam, como o evento `ev1` na transição de s_1 para s_2 no exemplo acima, e, opcionalmente, uma ou mais condições de guarda.

Além disso, as transições podem ter ações associadas. Como forma de habilitar diversas possibilidades ao usuário, as ações serão descritas como chamadas de funções, por conseguinte, a descrição da máquina torna-se mais sucinta sem reduzir a complexidade de suas atribuições, e sim o oposto, elevando-a.

Uma importante sugestão é que as funções empregadas como ações devem ser

desenvolvidas pelo usuário em um outro arquivo, como uma biblioteca. Assim sendo, é possível alterá-las de forma independente da máquina de estados, sem a necessidade de alterar o código desta, em caso de ajuste de alguma ação.

A sintaxe para as ações pode ser implementada acrescentando-se o símbolo de barra, após a declaração da transição. Por exemplo, caso a transição do estado s_1 para o estado s_2 vinculada ao evento `ev1` esteja relacionada à ação descrita pela função `f1()`, a definição dessa transição seria: `s1 -> s2 : ev1 / f1()`.

Baseado nas ideias mencionadas, e acrescentando-se funcionalidades como condições de guarda e transições internas e locais, chega-se à gramática desenvolvida neste trabalho, ilustrada na Figura 4.2.

```
root: (state | transition | initial_transition)*

state: "state" STATE "{" (state | transition | initial_transition
                | internal_transition | local_transition)* "}"

transition: (STATE)? "->" (STATE | ENDPOINT) ":"
                (TRIGGER ("," TRIGGER)*) ("[" GUARD "]")?
                ("/" BEHAVIOR)?

initial_transition: ENDPOINT "->" STATE ":" ("[" GUARD "]")?
                ("/" BEHAVIOR)?
internal_transition: ":" TRIGGER (("," TRIGGER)*)?
                ("[" GUARD "]")? ("/" BEHAVIOR)?

local_transition: (STATE)? "->"  "local" (STATE | ENDPOINT) ":"
                (TRIGGER ("," TRIGGER)*)? ("[" GUARD "]")?
                ("/" BEHAVIOR)?

STATE: CNAME
TRIGGER: CNAME
GUARD: ESCAPED_STRING
BEHAVIOR: ESCAPED_STRING
ENDPOINT: "[*]"
```

Figura 4.2: Gramática desenvolvida

Fonte: A autora.

Nesta gramática, pode-se ver a declaração de seis regras, `root`, `state`, `transition`,

`initial_transition`, `internal_transition` e `local_transition`; e a definição de cinco terminais, `STATE`, `TRIGGER`, `GUARD`, `BEHAVIOR` e `ENDPOINT`, todos de acordo com a sintaxe do Lark, abordadas no Capítulo 3. Esses elementos serão discutidos a seguir.

4.1.1 Os elementos terminais da gramática

Serão discutidos primeiro os elementos terminais da gramática, que são usados para compor os demais elementos.

Os dois primeiros terminais, `STATE` e `TRIGGER`, representam os estados e os eventos e são definidos como simples nomes alfanuméricos, sem espaços, usando a definição `CNAME` do Lark. Esta definição é bem semelhante à definição dos nomes de variáveis, na maioria das linguagens.

Os dois terminais seguintes, `GUARD` e `BEHAVIOR`, representam as condições de guarda e as ações. Como se quer que esses elementos possam conter linhas de código genéricas ou frases em uma linguagem natural, eles são definidos como *strings* alfanuméricas que podem conter espaços dentro delas.

Por último, é definido um terminal, `ENDPOINT`, para representar um estado inicial ou final, ou o estado-origem no caso da transição inicial de um superestado. Este terminal é a string fixa "[*]".

Os terminais são usados pelos outros elementos, as regras, para se formar sentenças que definem a máquina de estado.

4.1.2 As regras da gramática

As regras definidas especificam a sintaxe da linguagem, ou seja, para este trabalho, a relação entre os estados e os vários tipos de transição em uma máquina de estados hierárquica, e onde cada um deles pode ocorrer.

A primeira regra, *root*, especifica o que pode ser colocado no nível da raiz, ou base, do programa que define a máquina de estados, ou seja, o que não está contido por nenhuma outro elemento. Seguindo uma prática comum em gramáticas do

Lark, esta regra é definida em função de outras regras subsequentes, que definem os elementos de uma máquina de estado.

Como definido na seção 2.3, toda máquina de estados necessita de pelo menos um estado inicial. Dessa forma, uma função da regra *root* é permitir que o usuário determine quais são seus estados-filho e, através da transição inicial, qual é o estado inicial da máquina. A gramática desenvolvida permite, então, o programa definir um ou mais estados-filho, e uma ou mais transições iniciais, no topo do programa. Além disso, para prover flexibilidade, pode-se definir também, no topo do programa, transições entre estados.

Como mencionado na Seção 3.2.2, a gramática não define a *semântica* da definição, ou seja, se as regras especificadas fazem sentido lógico ou não. Por exemplo, a gramática não impede de se ter duas transições iniciais idênticas. As restrições de semântica são verificadas pela segunda parte do compilador desenvolvido, depois de feita a análise sintática.

Uma das restrições verificadas é se uma transição no topo do programa realmente define o estado de origem, uma vez que não é possível inferi-lo como no caso onde a transição é definida dentro de um estado. Além disso, se um dos estados não existir, ele é criado pelo compilador como se o programa o tivesse definido com a regra de estado. Se o estado que não existia for o de origem, a definição implícita deste estado conterá a transição processada.

Voltando às regras, a próxima regra é a *state*, que define a sintaxe da definição de estados, estabelecendo que deve iniciar com o literal *state*, seguido do nome do estado, representado pelo terminal `STATE`, sucedido por seus estados filhos e transições diversas, todos entre chaves, indicando que pertencem ao estado declarado naquela linha. Um estado pode conter zero ou mais de qualquer um destes itens, razão pela qual foi utilizado o operador gramatical *ou:* / entre os itens. O asterisco ao final do agrupamento entre parênteses indica que aquele grupo pode ocorrer zero ou mais vezes.

Devido aos vários tipos distintos de transições existentes, como visto na Seção 2.4,

há quatro regras distintas para elas: *transition*, *initial_transition*, *internal_transition* e *local_transition*. A diferença básica de sintaxe entre as regras está nos estados inicial ou final. A transição inicial não tem um estado inicial, e a regra denota isso especificando que os caracteres `[*]` devem estar como o estado inicial. No caso da regra da transição interna, a sintaxe é de que ambos os estados, inicial e final, devem estar ausentes.

Por outro lado, todas as transições, com a exceção da transição inicial, tem os elementos de evento, condição de guarda e ação como elementos em comum. A transição inicial não contém o evento. Deve-se notar que esses elementos são opcionais e não precisam necessariamente ser definidos quando do uso de uma regra. A exceção é o evento, que é elemento obrigatório, quando da definição de uma máquina de estados, usando esta gramática.

Para demonstrar o uso desta gramática, um exemplo é dado a seguir.

4.1.3 Exemplo do uso da gramática

A máquina M6, representada na Figura 4.3, é resultado do acréscimo de complexidade à máquina representada na Figura 2.5, incluindo transições, transição interna, eventos, condições de guarda e ações associadas a uma transição. Uma boa prática é envolver a máquina em um grande estado s_0, dessa forma, a transição inicial terá um estado definido.

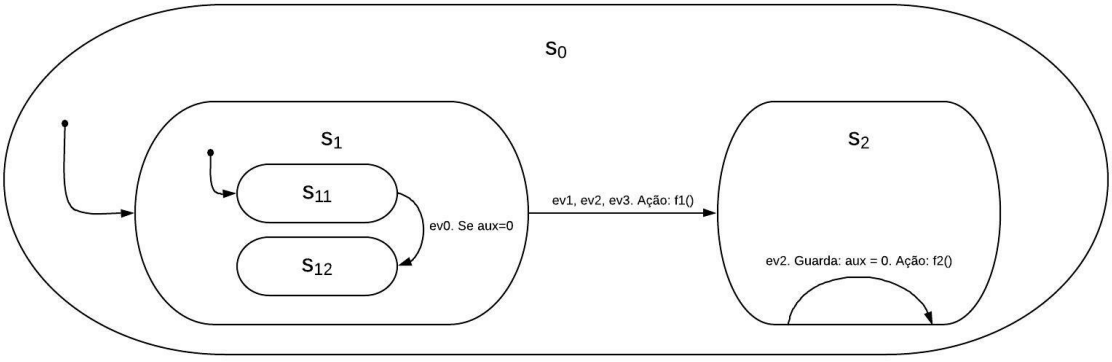

Figura 4.3: Diagrama da máquina de estados hierárquica M6

Fonte: A autora.

Na Figura 4.4, a declaração da máquina de estados M6, de acordo com a gramática desenvolvida. Vale ressaltar que há duas possibilidades de descrever uma transição, como demonstrado pelas transições de $S_{11} \rightarrow S_{12}$ e $S_1 \rightarrow S_2$, omitindo ou não o estado inicial da transição.

```
state S0 {
    [*] -> S1 :
    state S1 {
        [*] -> S11 :
        state S11 {
            S11 -> S12 : ev0 ["aux == 0"]
        }
        state S12 {
        }
        -> S2 : ev1, ev2, ev3 / "f1()"
    }
    state S2 {
        : ev2 ["aux = 0"] / "f2()"
    }
}
```

Figura 4.4: Declaração da máquina de estados M6

Fonte: A autora.

A partir desta declaração, é possível, para o analisador de sintaxe desenvolvido neste projeto, interpretar a máquina. Ou seja, retornar uma estrutura identificando todos os estados, seus subestados, ações, transições e tudo o mais que caracterize a máquina e seus estados. O desenvolvimento deste analisador é o tema da próxima seção.

4.2 Estrutura do compilador

Como mencionado na Seção 3.2.2, o Lark retorna a árvore sintática, e a partir desta é possível separar o conteúdo de cada terminal e regra declarados na gramática. Dessa forma, ao encontrar as palavras reservadas pela gramática desenvolvida, é possível discernir qual o tipo do elemento em questão, ou seja, se é um estado, uma transição interna, uma transição local, uma condição de guarda, entre outros.

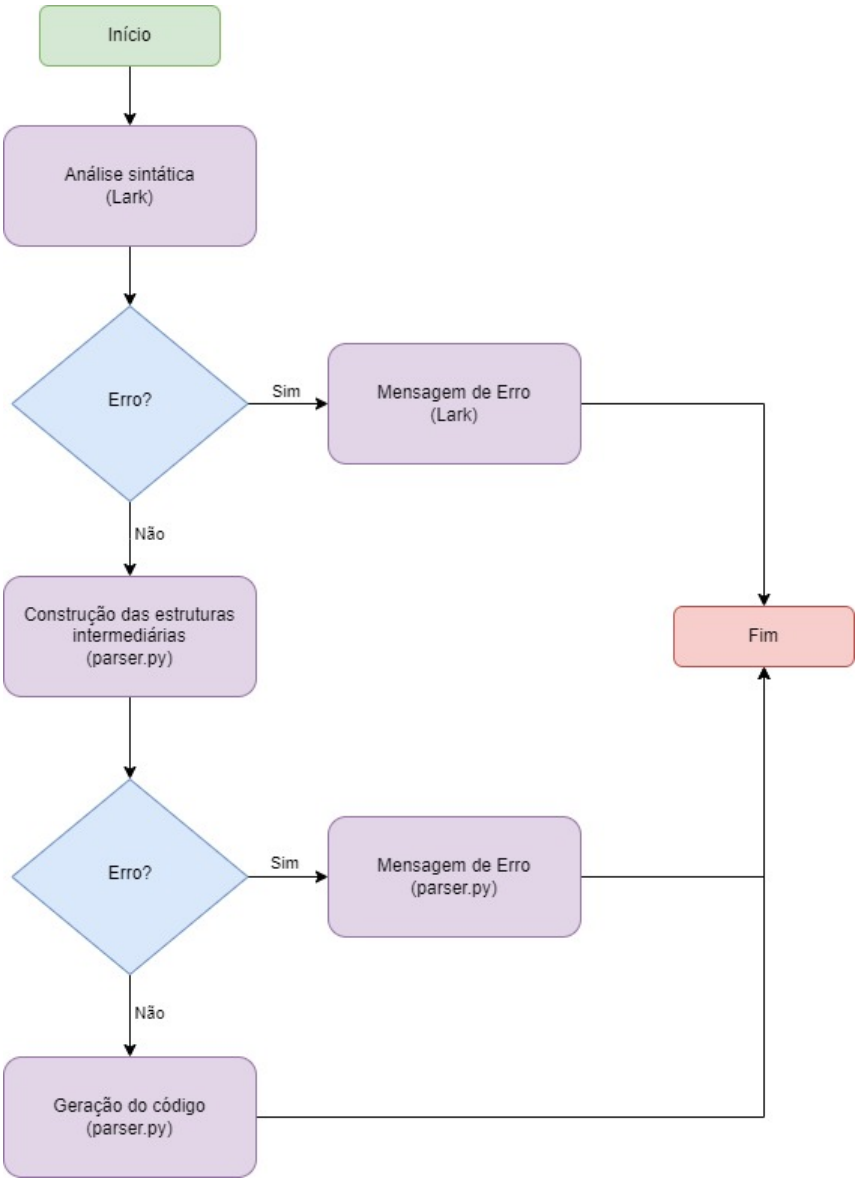

Figura 4.5: Etapas da compilação de uma máquina de estados

Fonte: A autora.

O compilador para geração do código da máquina de estados, faz uso do Lark para implementar as suas funcionalidades. A Figura 4.5, mostra o processo de compilação. A primeira etapa é o uso do Lark para geração da árvore sintática com as características já descritas. Com esta árvore pronta, segue as etapas de análise semântica e geração do código C, discutidas nas seções seguintes.

4.2.1 Análise semântica

Na análise semântica, a árvore sintática é percorrida e algumas estruturas de dados são construídas com o objetivo de facilitar o processo de geração de código. Essas estruturas de dados são voltadas para armazenar informações sobre os estados e sobre os vários tipos de transições.

A primeira estrutura escolhida é um dicionário, `state_dict`, definido inicialmente como `state_dict[state] = [{}, {}, {}, {}, []]`. A chave deste dicionário `state_dict` é o nome do estado `state`, e seu conteúdo é uma lista abrangendo cinco elementos, onde quatro deles são dicionários que serão responsáveis por armazenar as várias transições associadas ao estado, e o quinto é uma lista para guardar o nome de cada estado-filho. A função responsável por preencher este dicionário, `parse_states`, percorre a árvore sintática e coleta as informações de todos os estados referentes às transições e estados-filho.

A segunda estrutura de dados é um outro dicionário, declarado como `bottom_up_state_dict[state] = parent`. Nele, é armazenado o estado-pai de cada estado. Ao contrário dos estados-filho, onde um estado pode ter qualquer número, incluindo zero, de estados-filho, cada estado tem exatamente um estado-pai. Se o estado foi definido diretamente na regra *root* da gramática, o estado-pai é tomado como o estado especial, denotado pela string "[*]". Na Figura 4.6, encontra-se as etapas da função `parse_states`.

Com esses dois dicionários completos, é exequível a análise das mais variadas formas de transição. Neste segundo momento, a função responsável por esta tarefa, `parse_transitions`, se ocupa em buscar todas as transições na árvore sintática gerada pelo Lark, a fim de finalizar o preenchimento do dicionário `state_dict`. Na Figura 4.7, encontram-se as etapas da função `parse_transitions`.

Como relatado anteriormente, cada estado da máquina tem uma entrada em `state_dict`, que consiste de uma tupla, com quatro dicionários e uma lista. Os dicionários armazenam as transições, onde cada dicionário é dedicado a um tipo de transição. Essa especialização dos dicionários facilita a implementação da próxima

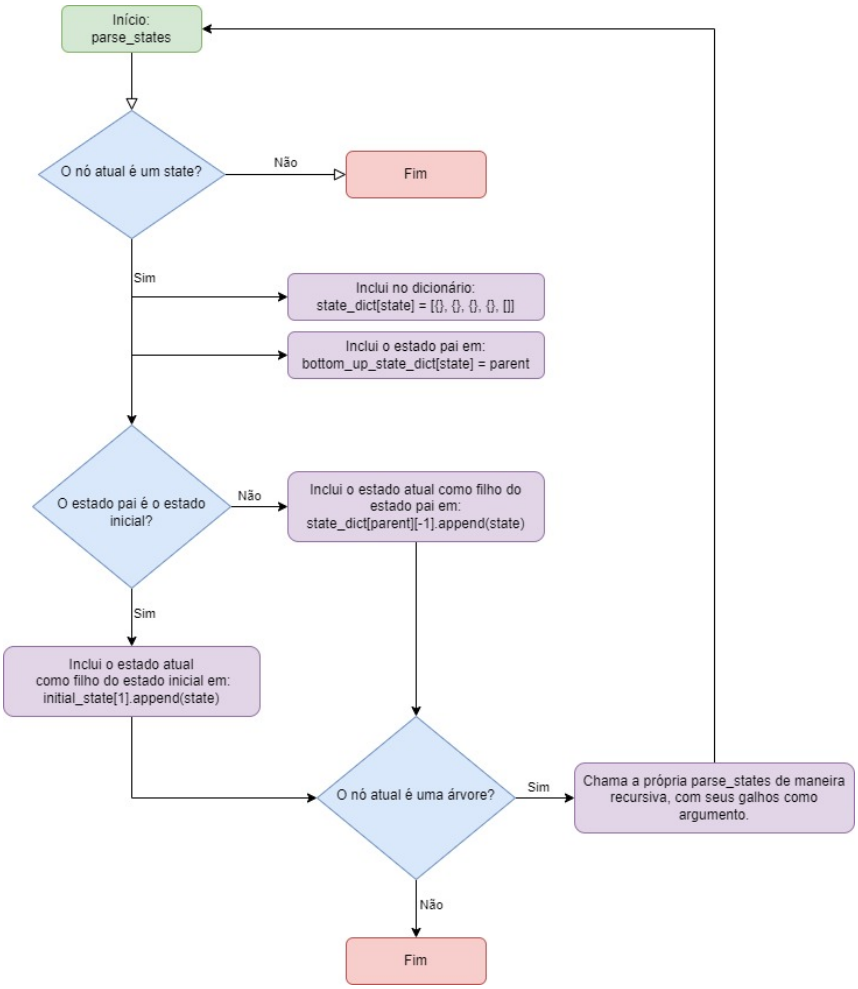

Figura 4.6: Etapas da função parse_states

Fonte: A autora.

etapa de geração do código C. Em ordem, os dicionários são para as transições iniciais, transições normais, transições locais e transições internas.

As entradas de cada dicionário são um pouco diferentes entre si. Para os dois dicionários relativos às transições normais e às transições locais, as entradas são criadas como:

```
transitions[(trigger, guard)] = (state2, behavior).
```

Neste caso, a chave do dicionário referente a uma transição é uma tupla, composta de seu gatilho e condição de guarda, onde a condição de guarda pode ser uma *string* vazia, no caso dela não ter sido definida. Observe que o gatilho nunca será uma *string* vazia, pois, se fosse, a análise sintática do Lark, apresentaria erro, uma vez que a gramática não permite essa possibilidade.

O valor associado a esta chave é uma tupla, consistindo do estado final da transição e da ação associada à transição. Observe que, assim como a condição de guarda, a *string* associada à ação, pode ser vazia, caso a ação não tenha sido definida.

A função responsável por tratar estes tipos de transição e complementar o dicionário é a `parse_external_local_tran`.

O caso de transições internas é ligeiramente diferente, com as entradas do dicionário correspondente, sendo criadas como:

```
transitions[(trigger, guard)] = behavior.
```

A diferença com relação aos tipos de transição expostos é que, por ser transição interna, não existe o estado final, uma vez que esse é o próprio estado que contém a transição, e assim o valor associado à chave, consiste de um simples valor, ao invés de uma tupla.

A função responsável por tratar as transições internas e complementar o dicionário é a `parse_internal_tran`.

Por último, para as transições iniciais, as entradas do respectivo dicionário são criadas como:

```
transitions[guard] = (state, behavior).
```

Observe que neste caso, a chave do dicionário é apenas a condição de guarda, uma vez que não há gatilho para a transição inicial. O valor associado à chave é uma tupla, consistindo do estado inicial, do estado em questão e da ação associada à transição. Vale ressaltar a obrigatoriedade de um superestado ter uma transição inicial sem condição de guarda, para que se garanta a inicialização do estado para algum de seus filhos. A função responsável por complementar este dicionário é a `parse_initial_tran`.

Além dos quatro dicionários descritos, viu-se que cada estado em `state_dict`, também tem uma lista associada a si, como o quinto elemento da tupla atribuída ao estado. Esta lista contém os estados-filho, do estado em questão. A lista é vazia se o estado não for um superestado.

Ao se utilizar os dois dicionários de estados, `state_dict` e `bottom_up_state_dict`, o programa desenvolvido neste projeto poderá gerar os arquivos necessários ao funcionamento da máquina de estados em linguagem C. As próximas seções versam justamente sobre esses arquivos.

É importante ressaltar que estes dois dicionários configuram uma representação intermediária da máquina, podendo ser utilizados para gerar a descrição da máquina em outras linguagens além de C, o que o torna versátil e útil para diversas aplicações.

4.3 Estrutura do código gerado

O compilador descrito na seção anterior gera, a partir de um texto descritivo de uma máquina de estados, de acordo com a gramática vista na Seção 4.1, um conjunto de arquivos C, contendo a implementação da máquina de estados. Estes arquivos podem ser compilados em separado para formar uma biblioteca, que pode ser posteriormente ligada (em inglês, *linked*) ao programa principal, que fará uso da máquina de estados. De forma alternativa, eles podem ser compilados junto com o programa principal.

Os arquivos gerados podem ser divididos em dois conjuntos: o primeiro consistindo dos arquivos que são comuns a todas as máquinas de estado, e que portanto não mudam com a máquina, e o segundo consistindo dos arquivos com a implementação dos estados, eventos e transições específicos da máquina descrita. Estes dois conjuntos de arquivos serão discutidos nas próximas duas seções. Nesta seção, será discutido como o código gerado está estruturado.

Do ponto de vista do programa principal, há três funções principais para tratamento dos eventos, contidas nestes arquivos: `wait_for_events()`, `dispatch_event()` e `set_event()`. Em geral, uma parte do programa (uma *thread*), lida com as duas primeiras funções, enquanto outra parte (seja interrupções, seja outras *threads*) usam a última função.

As duas últimas funções, `dispatch_event()` e `set_event()`, são complementares, no sentido que são usadas para inserir e recuperar, respectivamente, um evento

da fila de eventos a serem tratados. Deve-se ressaltar que esta fila de eventos é uma variável interna, com a qual o programa principal não deve se preocupar.

As duas primeiras funções, `wait_for_events()` e `dispatch_event()`, são geralmente usadas em conjunto, onde a primeira recupera um evento da fila e a outra o processa. O seguinte exemplo, ilustrado pela Figura 4.8, mostra onde o evento `EMPTY_EVENT` é reservado para sinalizar a parada do processamento de eventos.

```
while (1) {
    ev = wait_for_events();
    if (ev == EMPTY_EVENT)
        break;
    dispatch_event(ev);
```

Figura 4.8: Exemplo de uso do `EMPTY_EVENT`

Fonte: A autora.

Deve-se observar que a função `wait_for_events()` é bloqueante, ou seja, se não houver um evento disponível, a função suspende a execução até que um evento esteja disponível, quando a execução é resumida e, ao final, o evento ocorrido é retornado. A função `set_event()`, ao contrário, não bloqueia, mas sobreescreve o último evento do mesmo tipo, que não tenha ainda sido tratado. Em outras palavras, conceitualmente o evento é armazenado como uma *flag* binária.

Além das três funções descritas, uma quarta função, `init_machine()`, é importante para o programa principal, uma vez que é ela que inicializa a máquina de estados. O programa principal deve chamá-la antes de tratar qualquer evento, passando como parâmetro a função da transição inicial da máquina, em princípio definida na sua descrição textual. Após a execução desta função `init_machine()`, a máquina estará no estado indicado pela transição inicial, com a ação associada tendo sido executada.

O compilador gera várias outras funções relativas às transições e estados da máquina de estados, que implementam o comportamento da máquina conforme a descrição textual. Com exceção da transição inicial, contudo, o programa principal não precisa chamar diretamente nenhuma destas funções, pois as funções citadas se

encarregam de chamar as funções corretas em reação ao evento tratado. No caso da transição inicial, conforme descrito no parágrafo anterior, o programa deve passar a função associada como parâmetro para a função `init_machine()`.

Como descrito na próxima seção, o compilador também gera uma enumeração contendo os eventos definidos pela descrição da máquina de estados. Essas constantes definidas na enumeração, podem ser usadas no programa principal, nas chamadas da função `set_event()`, para tornar o programa mais legível.

4.4 Arquivos gerados

Nesta seção serão descritos os arquivos gerados pelo compilador, que implementam o funcionamento da máquina de estados hierárquica em linguagem C de forma personalizada, ou seja, que variam de acordo com a descrição da máquina. O primeiro dos arquivos gerados, o **hsm.c**, lida com os estados, transições, ações e condições de guarda, sem se preocupar com a questão hierárquica, função do segundo arquivo, o **transitions.h**.

O arquivo hsm.c é responsável por detalhar a máquina de estados com todas as suas características. Melhor dizendo, é onde se encontra a declaração de todos os estados e quais eventos cada um trata, como também as ações e condições de guarda referentes aos eventos.

Inicialmente são elencados, através de uma enumeração, todos os eventos que a máquina deve tratar. Essas informações são encontradas graças à lista `event_lst`, gerada a partir da leitura do dicionário `state_dict`.

Os eventos são enumerados a partir do valor da macro `USER_EVENT`, previamente atribuído no arquivo **sm.h**. Neste arquivo auxiliar pré-definido e necessário estruturalmente, descrito em mais detalhes na Seção 4.5, `USER_EVENT` é definido em uma enumeração como o quinto e último, dentre outros tipos de eventos reservados, dentre os quais: `EMPTY_EVENT`, `ENTRY_EVENT`, `EXIT_EVENT` e `INIT_EVENT`, com valores indo desde o zero até quatro. Dessa forma, para o programa, os eventos determinados pelo usuário terão valores a partir de quatro.

Esse valor não tem significado de prioridade, ou nada do tipo, sendo apenas uma maneira encontrada para que os eventos fossem tratados como números, computacionalmente falando, e não como *strings*, como declaradas pelo usuário, além de poder limitar sua quantidade.

Um total de 32 eventos podem ser utilizados neste desenvolvimento, incluindo os eventos já mencionados. Este número é determinado por uma outra macro, declarada como `MAX_EVENTS` em um segundo arquivo auxiliar, o **event.h**, também descrito na próxima seção. Além desta macro, o tipo de dado **event_t**, que define o *buffer* de eventos utilizados pelas funções `wait_for_events()` e `set_event()`, também pode restringir o número máximo de eventos. Neste trabalho, este tipo foi definido como **uint32_t**, e portanto suporta até 32 eventos. Se o programa final não precisar de mais do que 16 eventos, este tipo de dado pode ser redefinido como **uint16_t**, o que, em uma plataforma como o microcontrolador ATmega328p do Arduino, agiliza a busca por eventos ocorridos.

Com relação aos estados da máquina, cuja relação pode ser encontrada também no dicionário `state_dict`, cada estado da máquina terá uma função de *callback*, declarada no arquivo **hsm.h** e definida no arquivo **hsm.c** com a assinatura de uma função que recebe, como argumento, um evento do tipo **event_t** e retorna um dado do tipo **cb_status**. Como se verá adiante, existe uma variável **_p_state**, definida como um ponteiro, para uma dessas funções de estado, que armazena a função de *callback* do estado atual da máquina.

Estas funções de *callback* dos estados, são implementadas considerando o tratamento dos eventos que cada um deve atender. Isto significa que são incluídas também todas as outras características dos estados, como a análise da condição de guarda, as ações e transições que lhes concerne. Sempre utilizando do mesmo dicionário `state_dict`, em que se encontram armazenados dados valiosos sobre cada estado, esta parte do código é responsável por retratar o que cada estado deve fazer ao receber determinados eventos.

Empregando o *switch-case*, as funções dos estados conseguem tratar cada evento

de acordo com o que foi determinado pelo usuário, utilizando o evento como argumento. As condições de guarda, por sua vez, são incorporadas no *case* do evento que está associada, onde são testadas no corpo do *case* com o uso do *if-else*. O mesmo acontece com as ações e as transições.

Todas as transições da máquina de estados são definidas no arquivo **transitions.h**. Cada transição é definida como uma função, cujo nome é especificado pelo próprio compilador, seguindo o padrão **estadoinicial_estadofinal_tran()**. Como há a possibilidade de que se tenha mais de uma transição com os mesmos estados inicial e final, porém disparadas por eventos e/ou condições de guarda distintos, um indicador numérico pode ser acrescentado, como **estadoinicial_estadofinal_tran_1()** e **estadoinicial_estadofinal_tran_2()**.

Completando os casos de transições, há os casos de nomeação da transição inicial e da transição local, as quais o estado inicial é um superestado e o final é um de seus subestados. Nestas circunstâncias a nomenclatura será da seguinte forma, respectivamente, **superestado_init_subestado_tran()** e **superestado_local_subestado_tran()**.

Observe que as transições internas não implicam em mudança de estado, e portanto não precisam de uma função de transição. Todos os testes e efeitos de uma transição interna são realizados no corpo da função do estado correspondente, dentro do bloco do *case* relativo ao evento que disparou a transição.

Os arquivos nomeados como `guardandactions.h` e `guardandactions.c`, armazenam todas as ações e condições de guarda descritas na máquina de estados, para que sejam implementadas posteriormente. Ressalta-se que as condições de guarda serão avaliadas por um *if*, dessa maneira, devem retornar um valor nulo, para indicar a condição lógica falsa, ou um valor não-nulo (diferente de 0), para indicar a condição lógica verdadeira.

Observe que toda compilação da máquina de estados sobrescreverá os arquivos com as versões-padrão. Assim, se o usuário fizer alguma modificação em algum destes arquivos, recomenda-se renomeá-lo, para evitar perder o trabalho já feito.

Isto se aplica a todos os arquivos, mas é mais provável que aconteça apenas com o arquivo **guardandactions.c**.

Finalmente, o arquivo principal, o *main*, pode ser gerado de duas maneiras, uma voltada para o Linux ou outra para o microcontrolador ATmega328p. Dessa maneira, há duas possibilidades, o **main_atmega.cpp** ou o **main_linux.c**. A seção 4.6, descreve o passo a passo para a geração dos arquivos para cada opção.

Dessa forma, finaliza-se a descrição dos arquivos gerados, de forma personalizada para a máquina de estados hierárquica. Outros arquivos são gerados de forma padrão, pois são invariantes, independentemente da máquina a ser elaborada. Serão estes arquivos o objeto da próxima seção.

4.5 Arquivos auxiliares necessários ao funcionamento da máquina

Além dos arquivos mencionados na seção anterior, há também os arquivos de BSP, *Board Support Package*. A função destes arquivos é permitir a flexibilidade de utilizar a máquina de estados em diversas plataformas, como no Sistema Operacional Linux e em um microcontrolador AVR. Isto posto, no **bps.h** encontra-se a inclusão das opções de bibliotecas das arquiteturas que podem ser utilizadas.

Junto a este trabalho, há os arquivos de suporte para microcontrolador AVR e Linux, são eles **bsp_avr.h**, **bsp_linux.h** e **bsp_linux.c**. Estes possuem funções essenciais definidas para a execução do programa, que são utilizadas em momentos críticos. A função `enter_critical_region()`, informa ao controlador que não deve executar nenhuma outra tarefa a não ser a seguinte à chamada da função, e a função `leave_critical_region()` libera o controlador.

Os arquivos **event.h** e **event.c** merecem uma abordagem melhor do que apenas a da seção anterior, pois apresentam funções importantes para de fato colocar a máquina para funcionar. Eles são responsáveis pela gestão dos eventos, no que diz respeito a esperar por um evento e armazená-lo na variável que guarda os eventos

que ainda não foram tratados. As funções encarregadas por tais tarefas, como visto antes, são `wait_for_events()` e `set_event(ev)`.

Por fim, os arquivos **sm.c** e **sm.h**, incumbidos pela gestão dos estados ativos. Entende-se por estados ativos, todos aqueles que são superestados do estado atual, e o próprio estado atual. Para exemplificar, segundo a ilustração da máquina M5 na Figura 2.5, caso o estado atual seja o S_{12}, os estados ativos seriam o próprio S_{12} e os estados o qual seria subestado, o que, neste contexto, é apenas o estado S_1.

Define-se, então, um vetor chamado **__active__states** de tipo **cb__t** com tamanho determinado pela macro `MAX_ACTIVE_STATES`. O tipo **cb__t** é definido como um ponteiro para uma função que recebe, como argumento, um evento do tipo **event__t** e retorna um dado do tipo **cb__status**. O vetor **__active__states** armazena, em ordem, todos os estados ativos. Dessa maneira, há uma limitação de subestados aninhados determinada pela macro citada. Como forma de acompanhar o estado atual, é criado o ponteiro ***__p__state**, que aponta para a posição em **__active__states** que reflete o estado atual.

Tendo em vista a funcionalidade de gerência dos estados, nestes arquivos há definições de funções responsáveis por caminhar na árvore de estados da máquina e também por lançar um evento para o estado atual. Para lançar um determinado evento, é utilizada a função **dispatch__event(ev)**, que utiliza justamente o ponteiro ***__p__state**.

Salienta-se que a função **dispatch__event(ev)** verifica se o estado atual trata o evento lançado. Em caso negativo, a função encarrega-se de lançar o evento para seu estado pai, e assim sucessivamente, enquanto o evento não for tratado ou alcançar o estado ativo mais externo. Se porventura nenhum dos estados ativos tratar o evento, o estado atual volta a ser o que já era, e o evento é ignorado.

Ao caminhar pela máquina de estados, há as seguintes possibilidades: o próximo estado é um subestado, um superestado, ou um estado irmão do atual, ou seja, que está no mesmo nível, pois possuem o mesmo estado pai.

Para permitir estas substituições de estado atual, utiliza-se das funções: **push__-**

state(st), a qual **st** é um subestado do atual, este será salvo no campo seguinte de **_active_states** e será o novo endereço apontado por ***_p_state**; **pop_-state()**, utilizada para que ***_p_state** volte a apontar para o estado pai; **replace_state(st)**, utilizada quando o próximo estado atual **st** é irmão do atual; e, por fim, **exit_inner_states()**, utilizada para sair de todos os estados ativos, dessa forma, ***_p_state** volta a apontar para a raiz.

4.6 Gerando os arquivos finais

O atual trabalho reuniu todo o código redigido em um único arquivo, englobando tanto a gramática desenvolvida quanto a geração dos arquivos descritos neste capítulo. O arquivo nomeado como **parser.py** pode ser encontrado no repositório do github[1].

Para execução do programa é necessário que o Python esteja instalado na máquina, que o arquivo **parser.py** e o arquivo de texto com a descrição da máquina de estados **descricao_hsm.txt**, por exemplo, estejam na mesma pasta do computador. Lembrando que a descrição da máquina deve seguir a gramática definida neste capítulo.

Em seguida, deve-se abrir o terminal de comando do Linux neste diretório, e digitar o seguinte texto:

Para gerar arquivos para AVR: `python3 parser.py descricao_hsm.txt,avr`

Para gerar arquivos para Linux: `python3 parser.py descricao_hsm.txt,linux`

Em seguida, todos os arquivos serão gerados no mesmo diretório. Com o intuito de ilustrar todo o projeto, o capítulo seguinte apresenta uma aplicação prática do projeto.

[1]Para aceder ao repositório, acesse <https://github.com/angelicamuniz/HSM_Parser>.

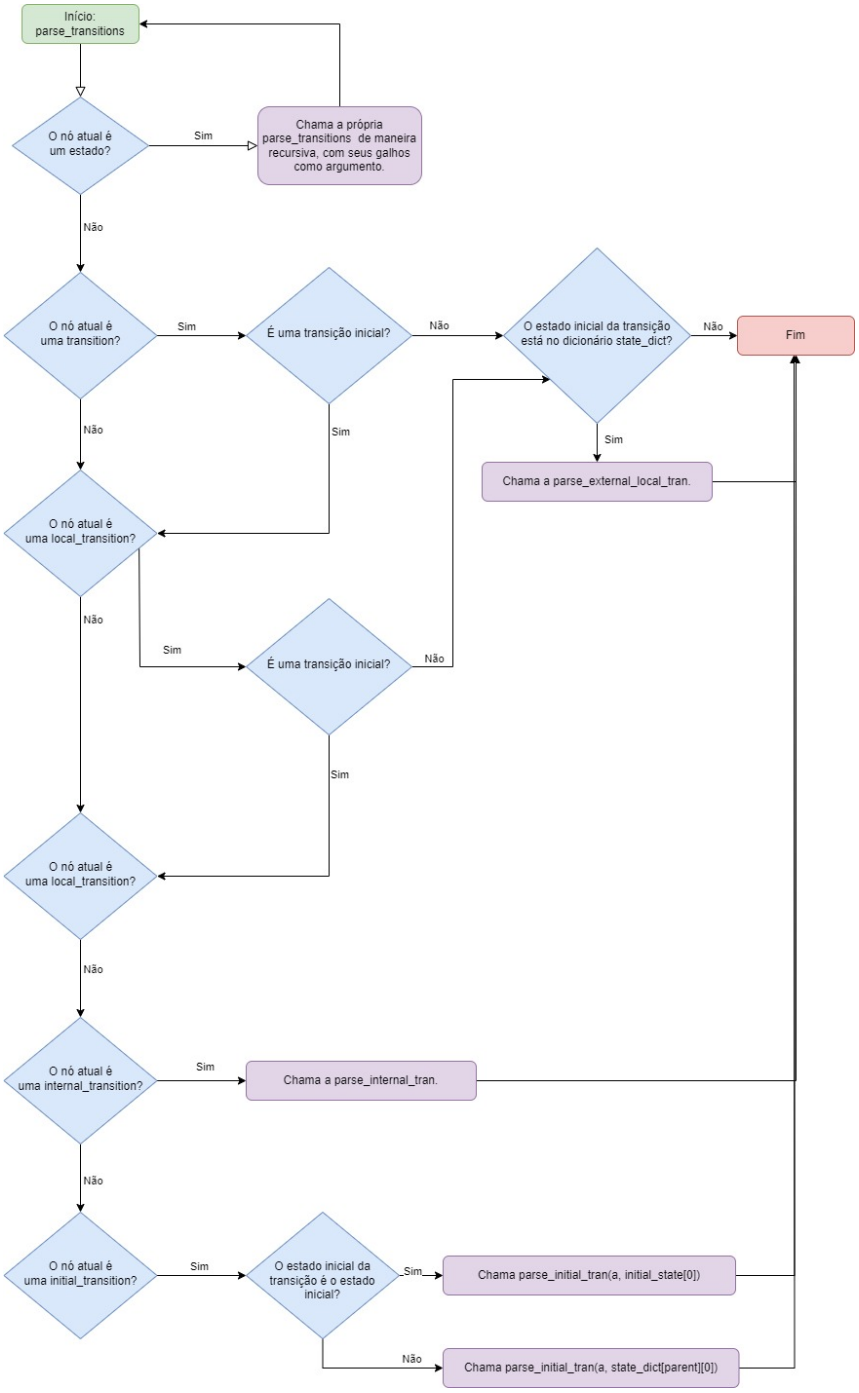

Figura 4.7: Etapas da função parse_transitions

Fonte: A autora.

Capítulo 5

Resultados

Para a apresentação dos resultados deste trabalho, foi elegido um projeto complexo o suficiente para implementar uma solução com máquina de estados hierárquica. Um sistema embarcado foi escolhido, sua placa de circuito impresso desenvolvida e montada para validação do *firmware* com a máquina de estados hierárquica, gerada a partir da DSL construída no projeto.

O projeto de um semáforo adaptável ao trânsito e aos pedestres, foi o desenhado para esta finalidade. Na Figura 5.1 está representado um cruzamento utilizando este sistema. O semáforo deve encontrar-se em um cruzamento de duas vias, uma principal e uma secundária. Complementando o semáforo, há sensores para pedestres que desejam atravessar a via principal, sensores para a presença de automóveis na via secundária, como também sensores para a presença de ambulância, em cada via.

O semáforo iniciará seu funcionamento com o verde para a via principal. Após um intervalo mínimo de tempo, deve verificar se há ambulância em uma das vias, e, em caso positivo, deve liberar o sentido em que a ambulância se encontra, até que esta finalize a passagem pelo cruzamento.

Supondo que não tenha a presença de ambulância, o sistema deve verificar se há pedestres para atravessar a via principal. Na condição afirmativa, deve então priorizar a passagem dos pedestres por um tempo determinado. Caso não tenha pedestres, finalmente poderá verificar se há a ocorrência de carros na via secundária, e assim liberar esta via por um tempo determinado.

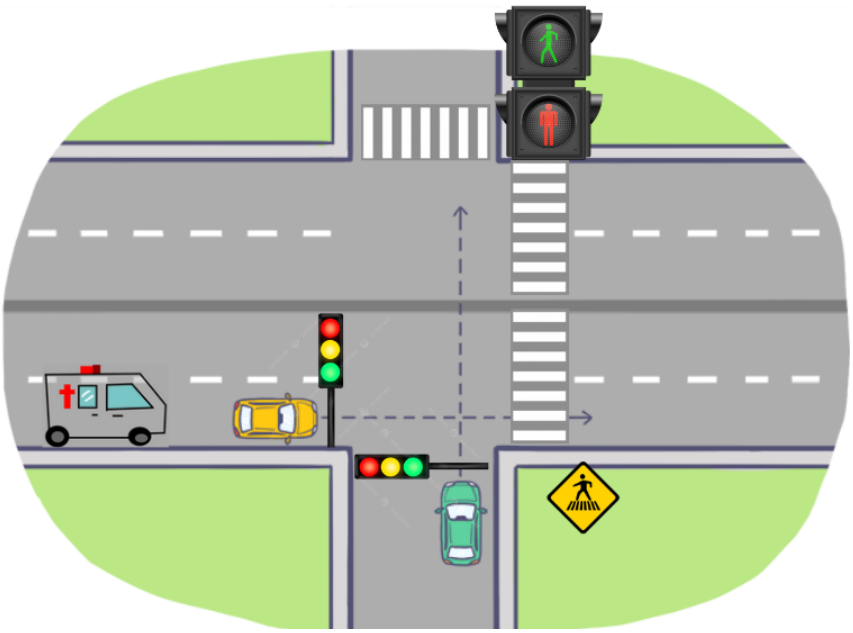

Figura 5.1: Ilustração do cruzamento com semáforos inteligentes

Fonte: A autora.

Em seu funcionamento, o semáforo da via secundária deve passar um tempo mínimo com o verde aceso. Em seguida, deverá verificar se há ambulância em sua via, ou na via principal, para que seja possível priorizar a passagem da ambulância. Em condição de ausência de ambulância, o sistema deve prontamente voltar o semáforo verde para a via principal, reiniciando o processo.

Detalhando um pouco mais o funcionamento do semáforo para os pedestres, quando este estiver ativo, deve ficar verde por um determinado intervalo de tempo, e depois piscar vermelho, alertando aos pedestres que o tempo da travessia já está prestes a acabar. Em seguida, verificar se há ambulância ou carros na via secundária, priorizando-os. Em caso de ausência de carros na via secundária ou de ambulâncias, voltar a preferência para a via principal.

O Modo Noturno complementa este semáforo inteligente. Quando acionado, os semáforos das duas vias ficam em amarelo piscante, como alerta para condutores ao atravessar a via.

Como auxílio visual, encontra-se o diagrama na Figura 5.2, com os estados, eventos, ações e transições considerados para o projeto.

O ATmega328p foi escolhido como o microcontrolador por ser largamente di-

fundindo como microcontrolador do Arduino Nano e Uno, apresentar uma extensa documentação e facilidade de compra. Para simular os sensores, foram elegidas chaves mecânicas. Mais detalhes da placa de circuito impresso encontram-se na Seção 5.3.

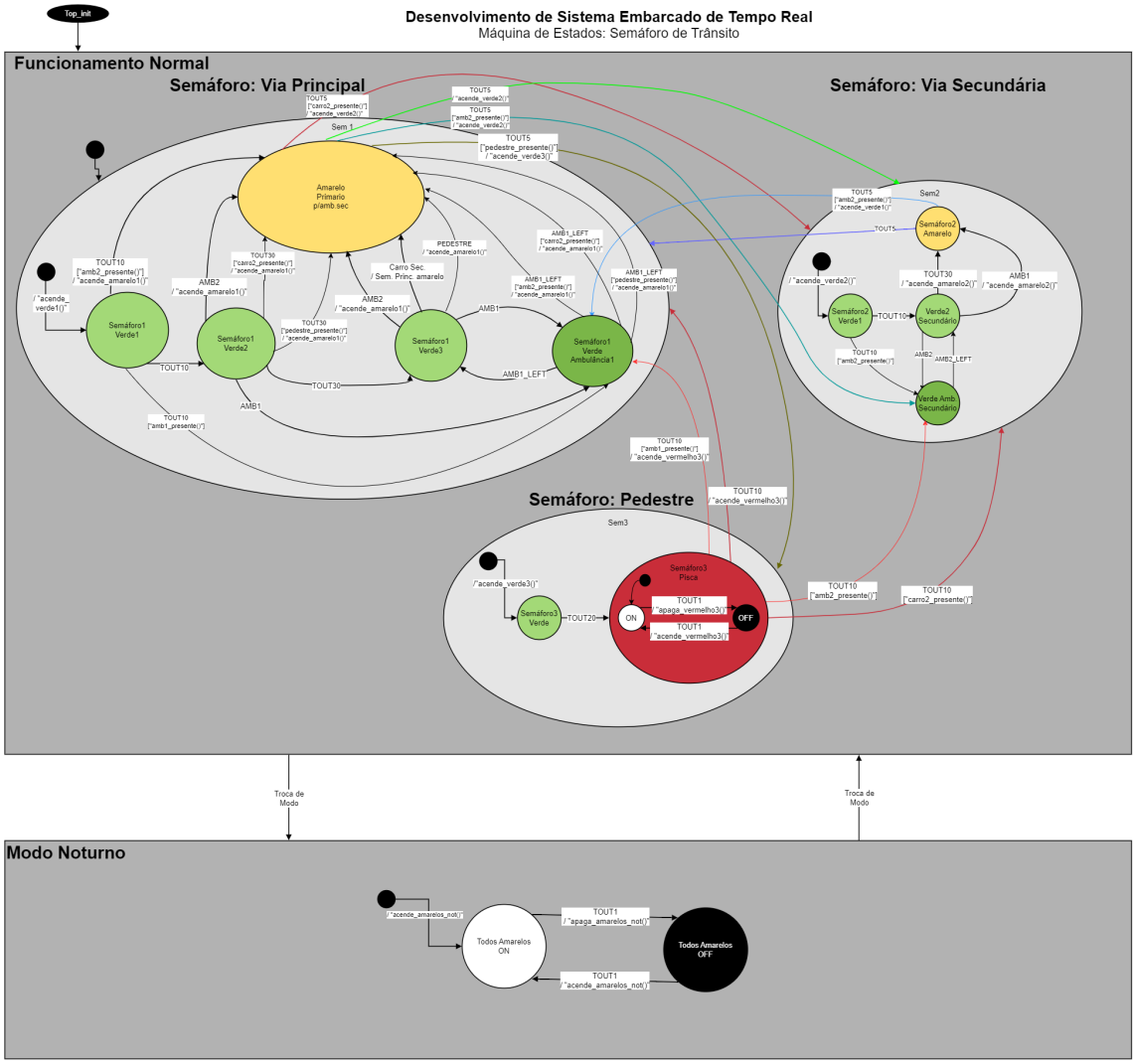

Figura 5.2: Diagrama da Máquina do Semáforo

Fonte: A autora.

5.1 Entrada do compilador

O texto de entrada do compilador deve seguir às orientações descritas na Seção 4.1. Dessa forma, a entrada foi redigida baseando-se no diagrama da Figura 5.2, considerando os estados, condições de guarda, ações e transições. O texto integral

utilizado como entrada do programa, encontra-se no github[1]. O seguinte trecho, na Figura 5.3, ilustra a descrição do super estado `Modo Noturno`.

```
state modonot {
        [*] -> modonot_amarelos_on : / "acende_amarelos_not()"
        -> fn : EVENT_TROCA_MODO

        state modonot_amarelos_on {
                -> modonot_amarelos_off : EVENT_TIMEOUT1 /
                "apaga_amarelos_not()"
        }

        state modonot_amarelos_off {
                -> modonot_amarelos_on : EVENT_TIMEOUT1 /
                "acende_amarelos_not()"
        }
}
```

Figura 5.3: Descrição do super estado `Modo Noturno`

Fonte: A autora.

O compilador implementado neste projeto, ao receber como entrada o código de descrição da máquina de estados do semáforo e a instrução de que o código gerado deve ser voltado para o microcontrolador AVR, gerará os arquivos necessários para a máquina de estados hierárquica, apresentados no Capítulo 4.

5.2 Saída do compilador

Os arquivos gerados a partir de uma entrada que descreve uma máquina de estados como orientado na Seção 4.1, voltados para o microcontrolador AVR, são os seguintes:

- bsp.h;
- bsp_avr.h;
- event.h;

[1]Para aceder ao repositório, acesse <https://github.com/angelicamuniz/HSM_Parser/tree/master/Exemplos/HSM-Semaforo>.

- event.c;
- guardandactions.h;
- guardandactions.cpp;
- hsm.h;
- hsm.cpp;
- main_atmega.ino;
- sm.h;
- sm.c;
- transitions.h.

O arquivo `guardandactions.h`, armazena todas as ações e condições de guarda descritas na máquina de estados, para que sejam implementadas posteriormente. O arquivo `transitions.h`, contém as informações necessárias para que a máquina percorra por todos os estados a cada transição. Os arquivos `sm`, `event` e `bsp` são padrão para qualquer projeto, dessa forma são iguais independente da máquina de entrada. A Seção 4.5 descreve cada um desses arquivos.

O arquivo `hsm.c`, compreende a forma da máquina de estados. São declarados todos os eventos, e as funções de `callback` são implementadas. Ao se basear no texto de entrada descrito na Seção 5.1, implementa cada estado adicionando suas transições, possíveis condições de guarda e ações.

Na Figura 5.4, encontra-se o trecho de código gerado pelo compilador para a entrada apresentada anteriormente, o super estado `Modo Noturno`. O arquivo completo também pode ser encontrado no mesmo repositório do *github*.

```
cb_status modonot_cb(event_t ev)
{
    switch(ev) {
    case ENTRY_EVENT:
        return EVENT_HANDLED;
    case EXIT_EVENT:
        return EVENT_HANDLED;
    case INIT_EVENT:
        acende_amarelos_not();
        modonot_init_modonot_amarelos_on_tran();
        return EVENT_HANDLED;
    case EVENT_TROCA_MODO:
        modonot_fn_tran();
        return EVENT_HANDLED;
    }
    return EVENT_NOT_HANDLED;
}

cb_status modonot_amarelos_on_cb(event_t ev)
{
    switch(ev) {
    case ENTRY_EVENT:
        return EVENT_HANDLED;
    case EXIT_EVENT:
        return EVENT_HANDLED;
    case EVENT_TIMEOUT6:
        apaga_amarelos_not();
        modonot_amarelos_on_modonot_amarelos_off_tran();
        return EVENT_HANDLED;
    }
    return EVENT_NOT_HANDLED;
}

cb_status modonot_amarelos_off_cb(event_t ev)
{
    switch(ev) {
    case ENTRY_EVENT:
        return EVENT_HANDLED;
    case EXIT_EVENT:
        return EVENT_HANDLED;
    case EVENT_TIMEOUT6:
        acende_amarelos_not();
        modonot_amarelos_off_modonot_amarelos_on_tran();
        return EVENT_HANDLED;
    }
    return EVENT_NOT_HANDLED;
}
```

Figura 5.4: Trecho de código gerado pelo compilador para o super estado `Modo Noturno`

Fonte: A autora.

Como pode-se observar, o compilador gerou corretamente tanto o arquivo `hsm.c` quanto os outros arquivos necessários para o funcionamento da máquina de estados hierárquica.

5.3 Desenvolvimento da placa de circuito impresso

A placa de circuito impresso para simular o semáforo foi desenhada com o programa DesignSpark PCB, um programa gratuito para design de placas. O diagrama de circuitos eletrônicos, comumente chamado de diagrama esquemático, ou simplesmente esquemático, é o primeiro passo para o desenvolvimento de uma placa. É nesta etapa em que os componentes são escolhidos e suas conexões são definidas.

Os semáforos da via principal, da via secundária e de pedestre foram representados por LEDs DIP, de forma que cada LED é controlado por um pino do microcontrolador, conectado à base de um transistor, operando como chave. Optou-se por isolar a corrente do LED do microcontrolador, como medida de proteção. A Figura 5.5 ilustra essa parte do esquemático.

Como representação dos sensores, foram utilizadas chaves, que conectam o pino que o microcontrolador realiza a leitura, ao terra ou à alimentação de 5V. Ou seja, dependendo da posição da chave, o nível lógico baixo ou alto é lido pelo microcontrolador.

Com o intuito de facilitar a visualização, um LED SMD foi conectado ao mesmo ponto em que o microcontrolador realiza a leitura, dessa forma, enquanto a chave estiver acionada, esse LED acenderá, representando a presença da ambulância, pedestre ou carro na via secundária. A Figura 5.6 ilustra essa parte do esquemático.

O microcontrolador ATmega328p e suas conexões para leitura das chaves e controle dos LEDs encontra-se ilustrado na Figura 5.7, assim como componentes importantes para seu funcionamento, como o cristal oscilador de 16 MHz, um filtro na entrada da alimentação e um botão para reinício forçado.

Finalizando, foi considerado que a placa já seria alimentada via conector micro USB por uma fonte de 5V DC, dessa forma, não se fez necessário inserir reguladores

de tensão, apenas um diodo TVS para suprimir picos de tensão e proteger contra danos causados por descarga eletrostática. Além disso, um conector para gravação do microcontrolador via SPI. A Figura 5.8 ilustra os dois circuitos.

O passo seguinte à criação do esquemático, é o design do *layout* da placa de circuito impresso. Por ser um circuito simples, uma placa com apenas duas camadas foi suficiente. Nas Figuras 5.9 e 5.10 encontram-se as camadas superior e inferior da placa, respectivamente.

Para a disposição dos componentes, foram consideradas as chaves que representam as ambulâncias nas parte superior das laterais, sendo a da Ambulância da via principal na lateral esquerda e a da via secundária na lateral direita. Na parte inferior da lateral direita encontra-se também a chave para representar o Carro na Via Secundária. Por último, na lateral inferior as chaves para representar o sensor de Pedestre, e para o acionamento do Modo Noturno.

Os LEDs DIP de cada semáforo foram distribuídos próximos à sua chave correspondente. Ademais, foram acrescidos quatro furos para fixação de suportes, para que a placa fique suspensa e não tenha contato com a superfície da mesa.

A confecção da placa foi encomendada à empresa chinesa JLCPCB. Fotos das camadas superior e inferior estão representadas pelas Figuras 5.11 e 5.12, respectivamente.

Em seguida, para soldar os componentes, foi utilizada a estação de solda Hikari HK-936A e solda em fio da marca Cobix. As Figuras 5.13 e 5.14 mostram o resultado da placa montada.

Como exemplo de funcionamento, duas fotos ilustram os seguintes momentos: o semáforo da via principal aberto, e os outros fechados, Figura 5.15; e o semáforo de pedestre aberto, em que pode-se observar a posição da chave indicando a presença do pedestre, e os demais semáforos fechados, Figura 5.16.

Para atestar o funcionamento da placa e da máquina de estados gerada pelo compilador, foi criado um plano de testes. Este plano será descrito na Seção 5.4.

5.4 Plano de testes

Uma seleção de 15 testes foram escolhidos para comprovar o funcionamento da máquina de estados e do *hardware* desenvolvido. Cada teste é composto por três partes:

Precondições: como o sistema encontra-se antes do passo seguinte;

Entrada: passo a ser realizado, como mudar o estado de alguma das chaves;

Saída: o que se espera após ser realizado o passo anterior.

A seguir, a descrição de cada teste:

- Teste 1:

  Precondições: semáforo iniciado há mais de 40 segundos, aberto para a via principal, e todas as chaves desligadas;

  Entrada: acionar chave que representa carro na via secundária;

  Saída: o semáforo da via principal acende amarelo para que após 5 segundos acenda vermelho, enquanto que o da via secundária acende verde. Após 40 segundos de preferência para a via secundária, o semáforo da via secundária acende amarelo para que após 5 segundos acenda vermelho, enquanto que o da via principal acende finalmente verde novamente.

- Teste 2:

  Precondições: semáforo iniciado há menos de 40 segundos, aberto para a via principal, e todas as chaves desligadas;

  Entrada: acionar chave que representa carro na via secundária;

  Saída: o semáforo da via principal acende amarelo somente depois que completar os 40 segundos desde que iniciou o verde da via principal, e então se repete o mesmo para a saída do Teste 1.

- Teste 3:

  Precondições: semáforo iniciado há mais de 10 segundos, aberto para a via principal, e todas as chaves desligadas;

  Entrada: acionar chave que representa ambulância na via secundária;

  Saída: o semáforo da via principal acende amarelo para que após 5 segundos acenda vermelho, enquanto que o da via secundária acende verde.

- Teste 4:

  Precondições: semáforo iniciado há menos de 10 segundos, aberto para a via principal, e todas as chaves desligadas;

  Entrada: acionar chave que representa ambulância na via secundária;

  Saída: o semáforo da via principal acende amarelo somente depois que completar os 10 segundos desde que entrou neste estado do "Semáforo Principal", e então se repete o mesmo para a saída do Teste 3.

- Teste 5:

  Precondições: semáforo iniciado há mais de 40 segundos, aberto para a via principal, e todas as chaves desligadas;

  Entrada: acionar chave que representa o pedestre;

  Saída: o semáforo da via principal acende amarelo para que após 5 segundos acenda vermelho, enquanto que o do pedestre acende verde. Após 20 segundos de preferência para o pedestre, este semáforo pisca vermelho por 10 segundos, e então o da via principal acende finalmente verde novamente.

- Teste 6:

  Precondições: semáforo de pedestres aceso em verde;

  Entrada: acionar chave que representa carro na via secundária;

Saída: após os 20 segundos de preferência para o pedestre, este semáforo pisca vermelho por 10 segundos, e então o da via secundária acende verde.

- Teste 7:

  Precondições: semáforo de pedestres aceso em verde;

  Entrada: acionar chave que representa a ambulância na via secundária;

  Saída: após os 20 segundos de preferência para o pedestre, este semáforo pisca vermelho por 10 segundos, e então o da via secundária acende verde.

- Teste 8:

  Precondições: semáforo de pedestres aceso em verde;

  Entrada: acionar chave que representa a ambulância na via principal;

  Saída: após os 20 segundos de preferência para o pedestre, este semáforo pisca vermelho por 10 segundos, e então o da via principal acende verde.

- Teste 9:

  Precondições: semáforo da via principal aceso em verde por ter detectado uma ambulância nesta via, com a chave desta ambulância acionada;

  Entrada: acionar chave que representa o carro na via secundária;

  Saída: o semáforo da via principal continua em verde.

- Teste 10:

  Precondições: semáforo da via principal aceso em verde por ter detectado uma ambulância nesta via, com as chaves da ambulância da via principal e carro na via secundária acionadas;

  Entrada: desativar a chave que representa a ambulância na via principal;

  Saída: o semáforo da via principal acende amarelo. Após 5 segundos acende vermelho, enquanto que o da via secundária acende verde.

- Teste 11:

  Precondições: semáforo da via principal aceso em verde por ter detectado uma ambulância nesta via, com a chave desta ambulância acionada;

  Entrada: acionar chave que representa a ambulância na via secundária;

  Saída: o semáforo da via principal continua em verde.

- Teste 12:

  Precondições: semáforo da via principal aceso em verde por ter detectado uma ambulância nesta via, com as chaves da ambulância da via principal e secundária acionadas;

  Entrada: desativar a chave que representa a ambulância na via principal;

  Saída: o semáforo da via principal acende amarelo. Após 5 segundos acende vermelho, enquanto que o da via secundária acende verde, dando preferência à ambulância.

- Teste 13:

  Precondições: semáforo da via secundária aceso em verde por ter detectado ambulância nesta via;

  Entrada: desligar a chave que representa a ambulância na via secundária;

  Saída: o semáforo da via secundária acende amarelo. Após 5 segundos acende vermelho, enquanto que o da via principal acende verde novamente.

- Teste 14:

  Precondições: chave do modo noturno desativada, independente de qual semáforo esteja com prioridade;

  Entrada: ativar a chave do modo noturno;

  Saída: os LEDs amarelos, da via principal e secundária, começam a piscar a uma frequência de 1Hz.

- Teste 15:

  Precondições: chave do modo noturno ativada;

  Entrada: desativar a chave do modo noturno;

  Saída: o semáforo reinicia seu funcionamento normal, acendendo verde para a via principal.

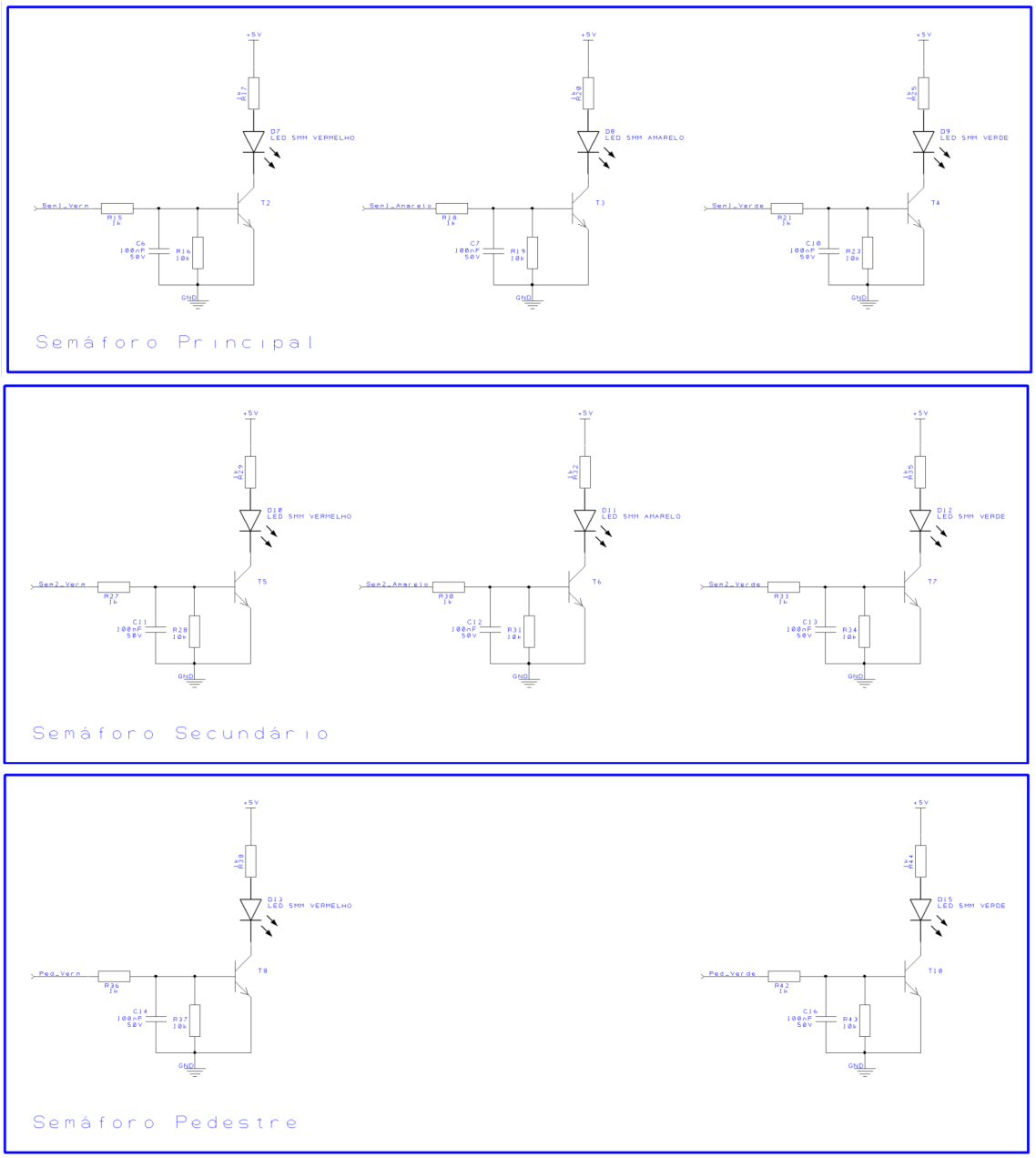

Figura 5.5: Esquemático do Semáforo - Seção dos LEDs

Fonte: A autora.

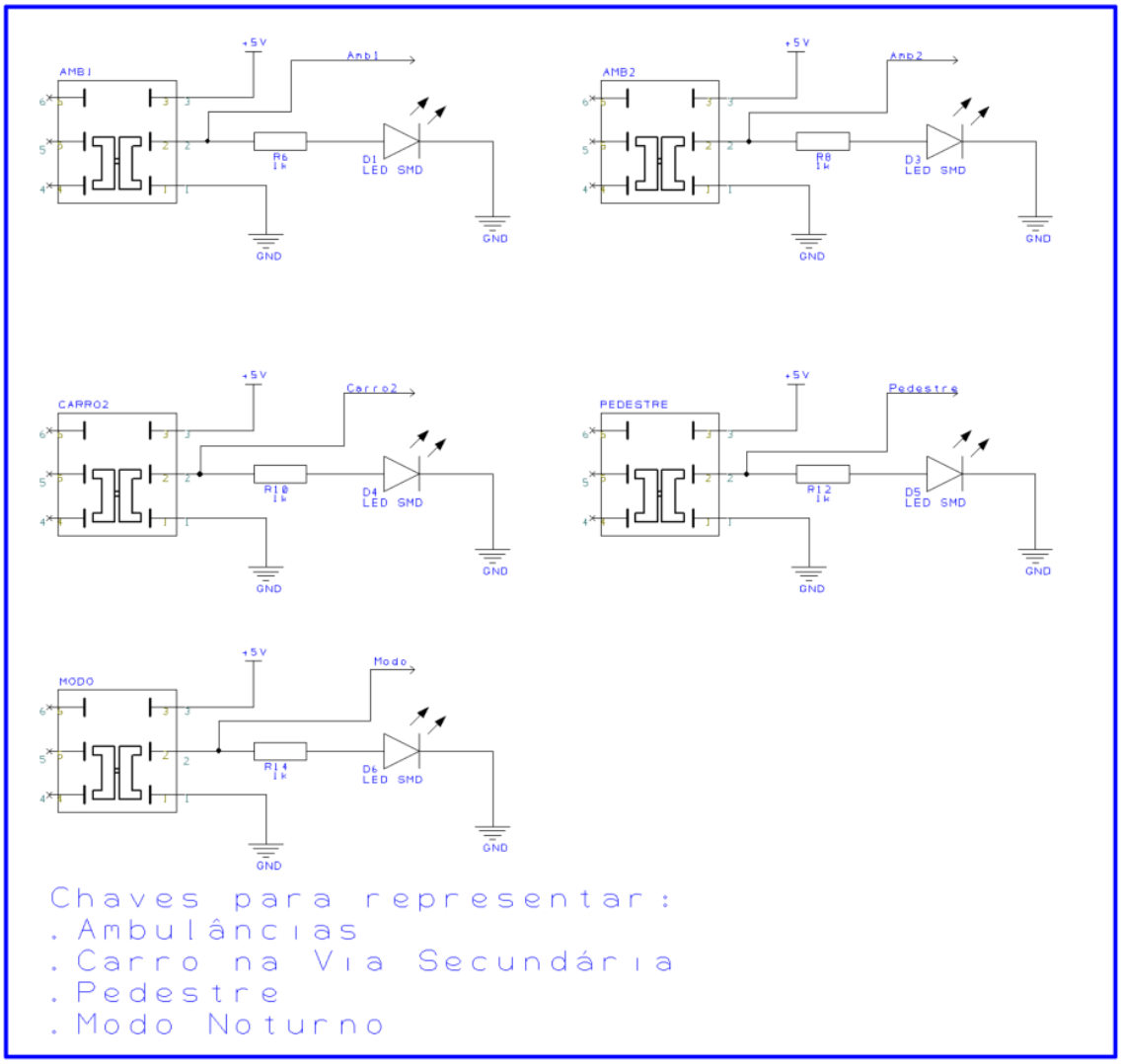

Figura 5.6: Esquemático do Semáforo - Seção das chaves como sensores

Fonte: A autora.

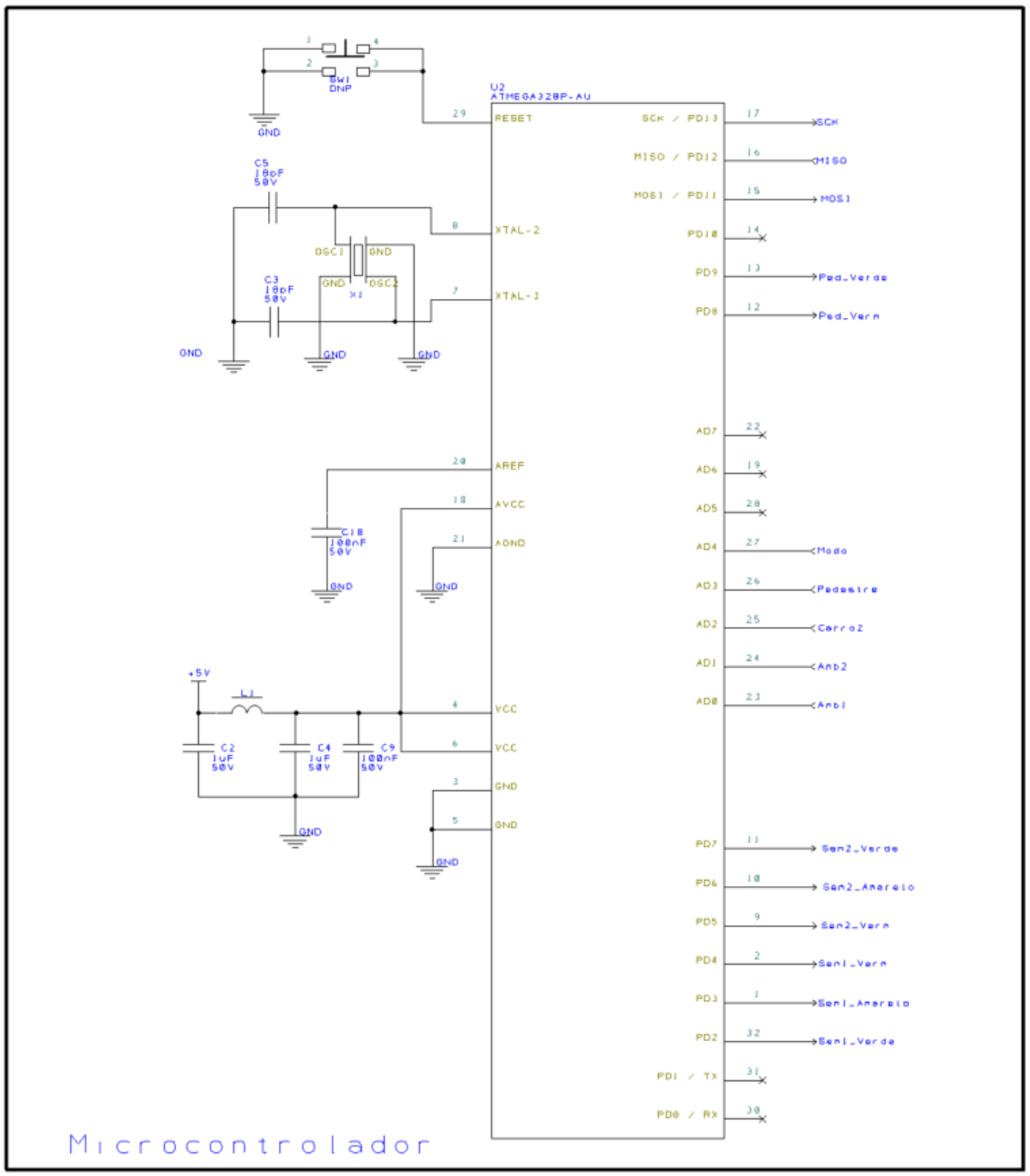

Figura 5.7: Esquemático do Semáforo - Seção do microcontrolador

Fonte: A autora.

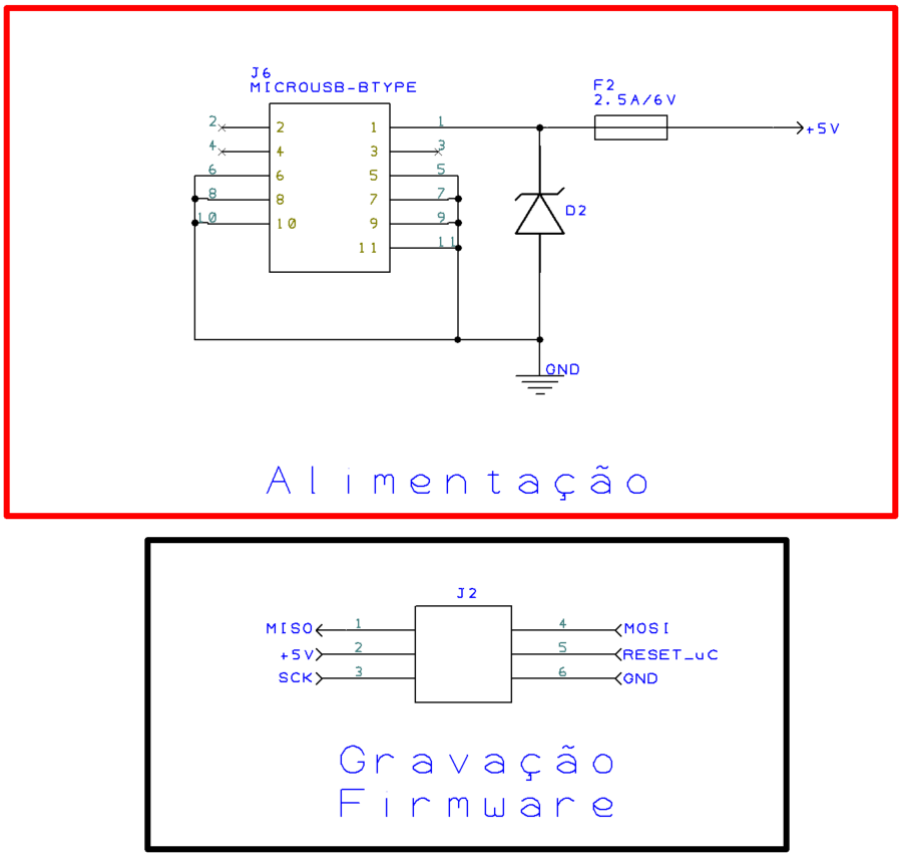

Figura 5.8: Esquemático do Semáforo - Seção da alimentação da placa e conector para gravação do *firmware*

Fonte: A autora.

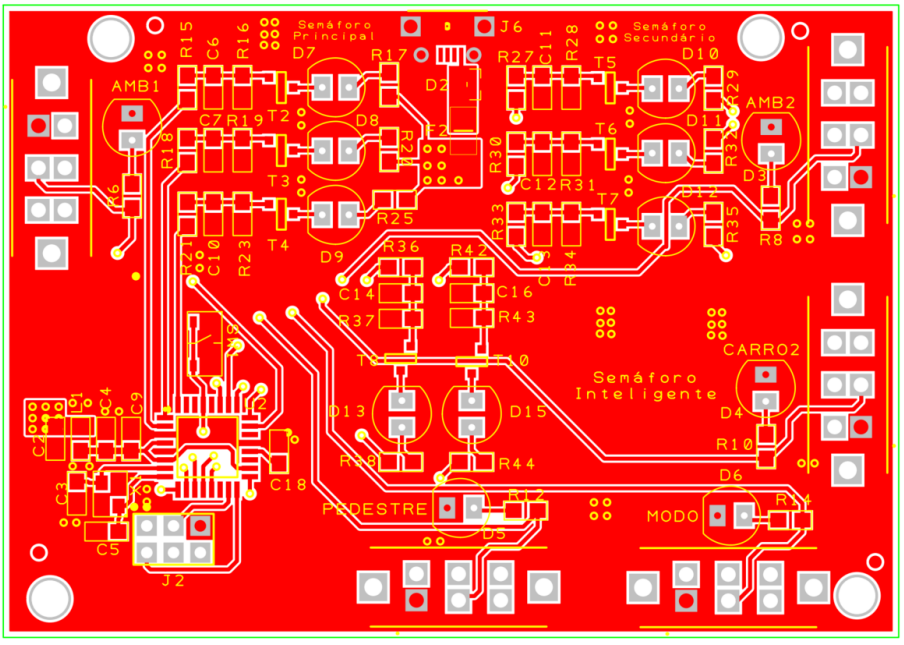

Figura 5.9: *Layout* do Semáforo - Camada Superior

Fonte: A autora.

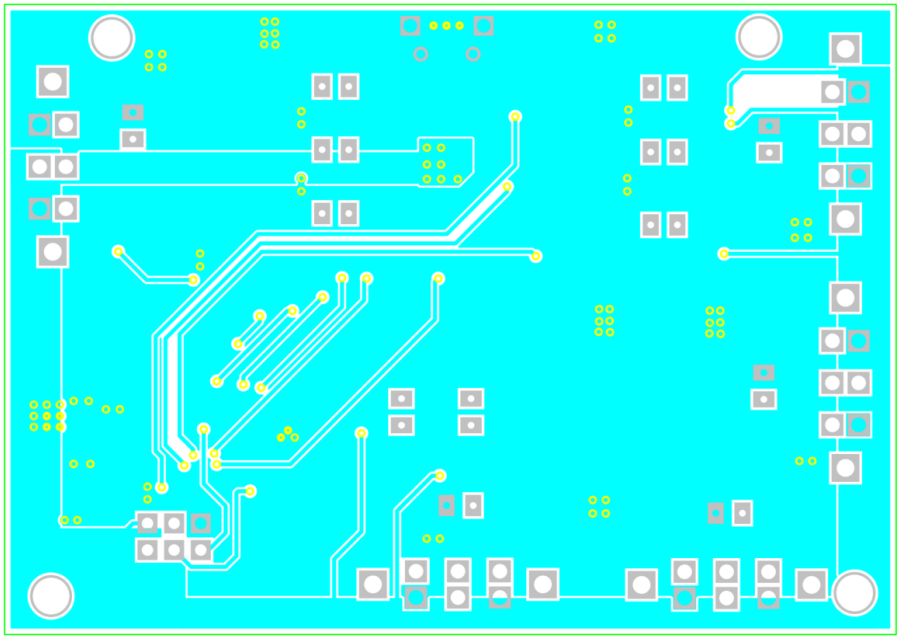

Figura 5.10: *Layout* do Semáforo - Camada Inferior

Fonte: A autora.

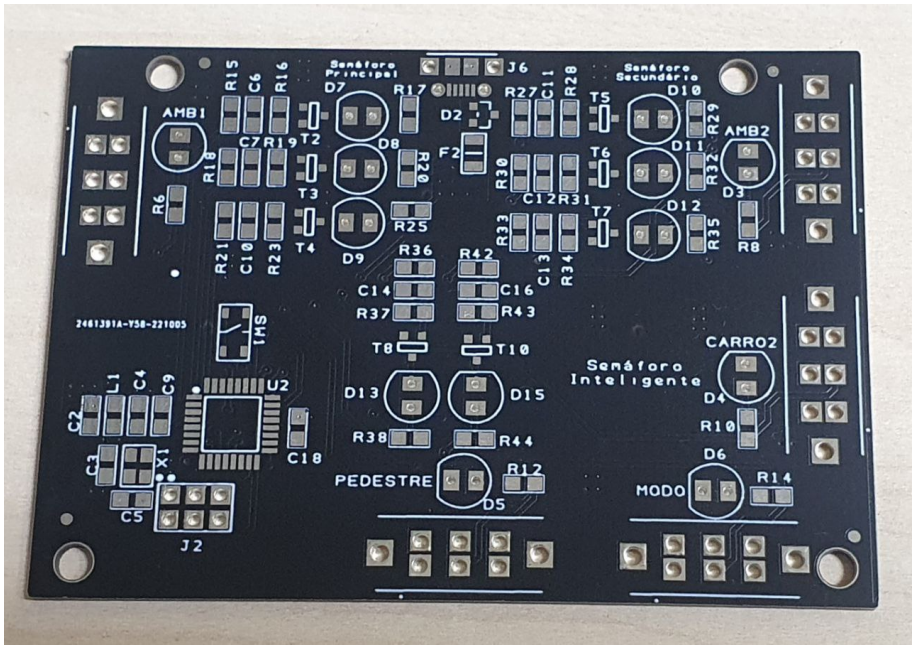

Figura 5.11: Placa do Semáforo - Camada Superior

Fonte: A autora.

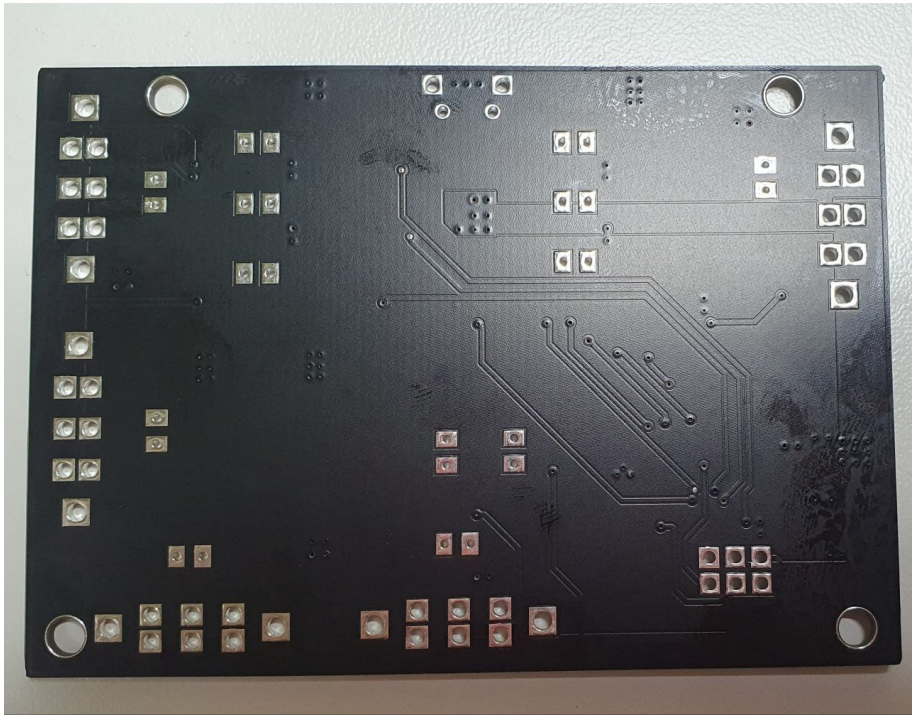

Figura 5.12: Placa do Semáforo - Camada Inferior

Fonte: A autora.

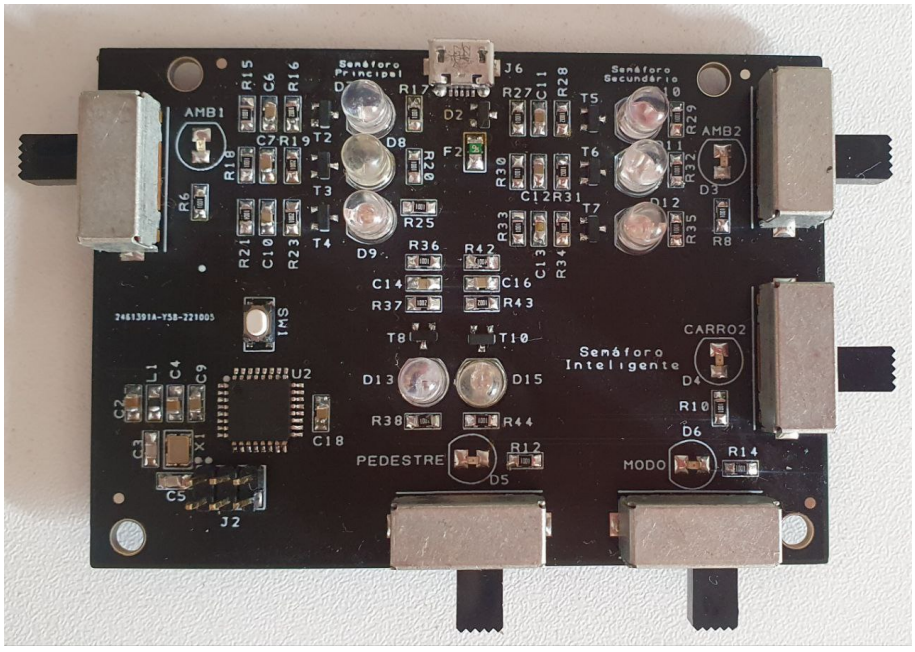

Figura 5.13: Placa do Semáforo Montada Camada Superior

Fonte: A autora.

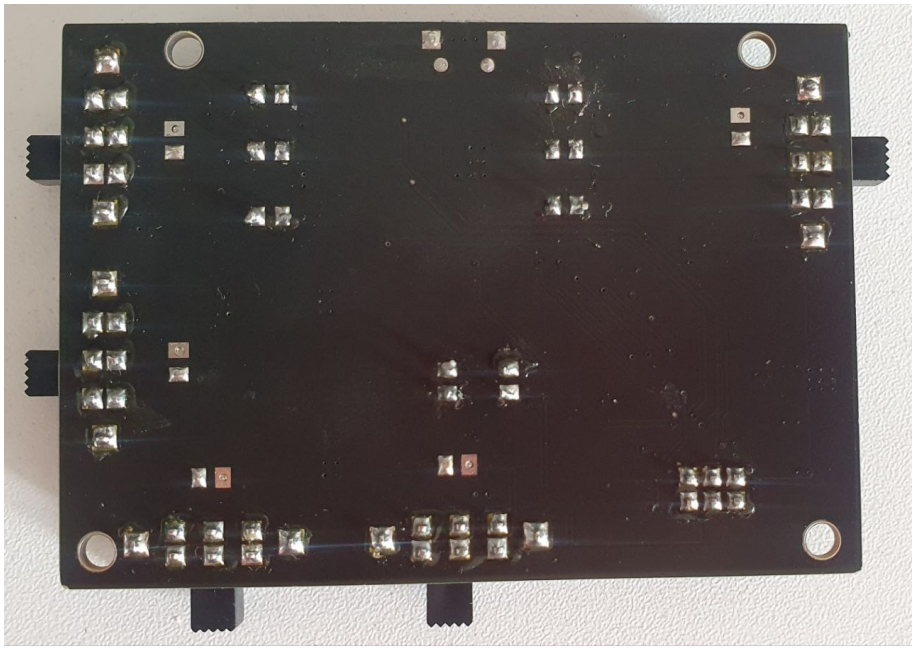

Figura 5.14: Placa do Semáforo Montada Camada Inferior

Fonte: A autora.

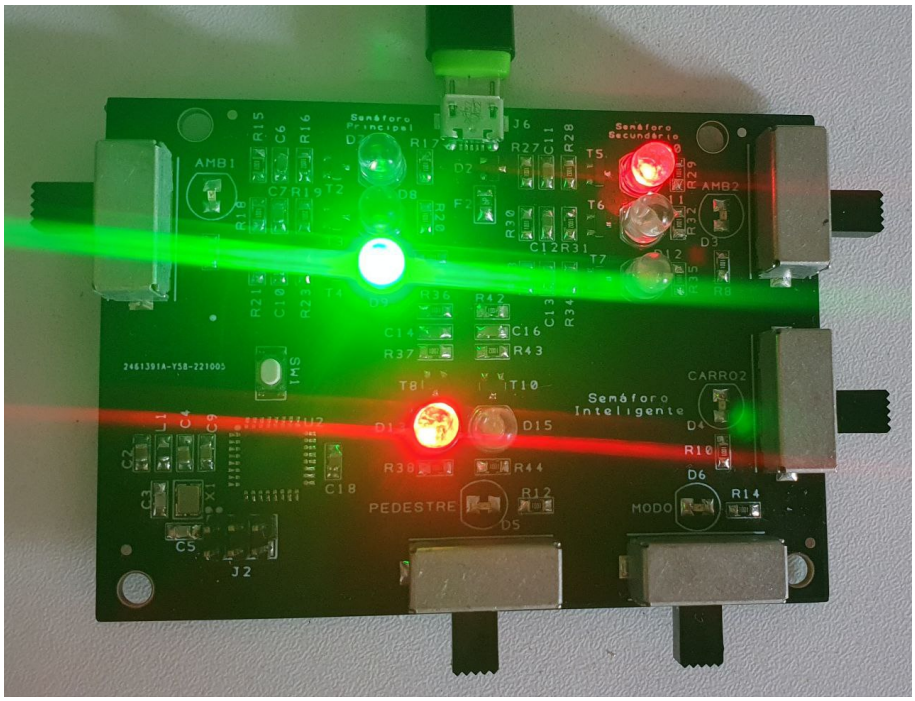

Figura 5.15: Semáforo em funcionamento - Via Principal

Fonte: A autora.

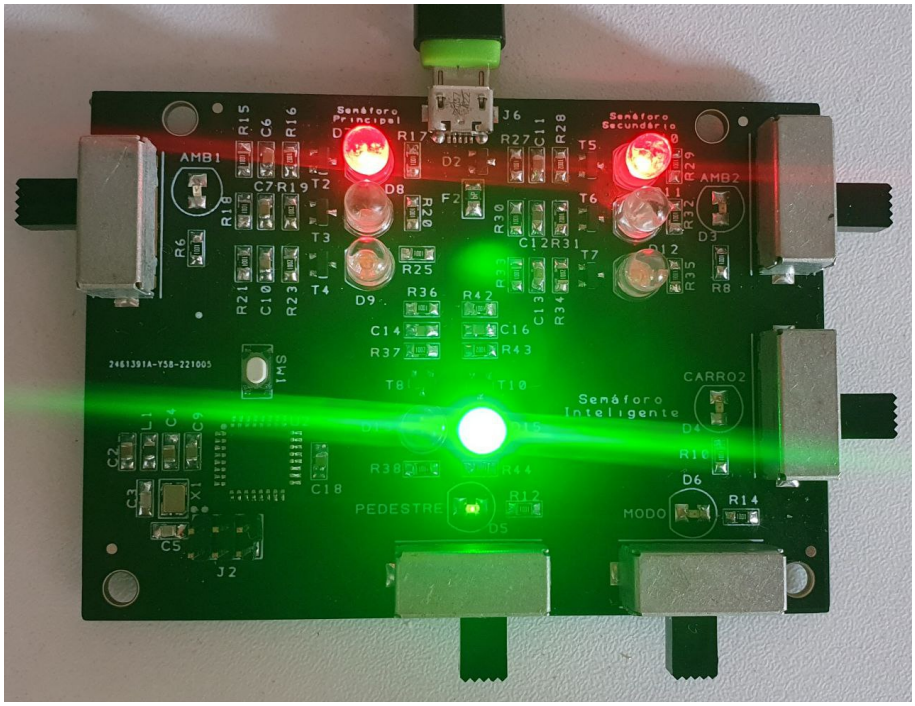

Figura 5.16: Semáforo em funcionamento - Pedestre

Fonte: A autora.

Capítulo 6

Considerações Finais

Os sistemas embarcados são cada vez mais utilizados em diversas áreas, desde sistemas de controle de automação até em dispositivos médicos, automóveis e aparelhos eletrônicos. Esses sistemas são projetados para executar funções específicas, em tempo real e com baixo consumo de energia. Levando em consideração que os sistemas embarcados têm caráter de ser reativo a eventos, a utilização de máquinas de estados hierárquicas se torna essencial, pois permite uma melhor organização e controle do fluxo de dados.

Tendo em mente o desafio do desenvolvimento de *softwares* robustos e confiáveis utilizando máquinas de estados e os paradigmas da programação reativa e orientada a eventos, este trabalho apresentou uma ferramenta de *software* que gera o código C necessário para execução de uma máquina de estados hierárquica a partir de sua descrição textual. O trabalho mostrou o potencial de uma Linguagem de Domínio Específico, também conhecido como DSL, para esse tipo de abordagem de descrição textual de um conceito.

Além disso, o uso de Python e sua biblioteca Lark mostrou que a implementação de um compilador para uma DSL não é tarefa muito difícil, e este trabalho fez bom uso dessa versatilidade. Espera-se que isto inspire o uso de DSL em outras aplicações.

O presente projeto se apresentou eficiente em seu propósito de fornecer uma potente ferramenta para a criação de *softwares* mais confiáveis, um diferencial im-

portante principalmente para sistemas complexos. A facilidade da linguagem de entrada criada e a versatilidade das máquinas de estados hierárquicas proporcionam ao desenvolvedor de sistemas embarcados um ganho de tempo, ao passo que torna o projeto seguro e estável.

Assim, espera-se que o trabalho elaborado seja de grande utilidade e aplicabilidade por entregar ao usuário a estrutura de uma máquina de estados hierárquica em C, a linguagem mais utilizada para a programação de microcontroladores, poupando-o do esforço de escrever e manter o código de gerenciamento da máquina de estados e permitindo-o focar nas ações de sua aplicação.

6.1 Dificuldades encontradas

Durante a execução deste trabalho foram encontrados desafios, particularmente devido ao contexto da pandemia que impôs limitações ao acesso a recursos e ambientes propícios ao desenvolvimento do projeto. A realização de parte do trabalho de forma remota exigiu adaptações por se dar distante do ambiente da Universidade.

Ademais, a produção da placa de circuito impresso também se mostrou mais uma dificuldade, tendo em vista a necessidade de fabricação e aquisição de componentes na China. Esta escolha foi tomada frente à dificuldade e ao custo elevado encontrados no Brasil. Esses fatores contribuíram para um processo mais demorado, devido a atrasos logísticos naturais da importação.

6.2 Trabalhos futuros

O trabalho atual foi desenvolvido para gerar a saída em linguagem C, por ser uma das mais utilizadas na programação de microcontroladores. Uma opção interessante seria expandir o projeto para gerar máquinas de estados hierárquicas em outras linguagens, como em Python ou Java, uma vez que máquinas de estados podem ser utilizadas em outros ambientes.

Outra melhoria a ser acrescentada é a adequação da máquina de estados a todos

os conceitos definidos pela OMG, para as máquinas de estados hierárquicas, como o conceito de máquinas de estados concorrentes e de memória, por exemplo. Este último conceito se refere a característica onde o sistema, ao percorrer os estados, crie um histórico e saiba por onde passou e assim possa retornar, quando necessário e programado, a um estado anterior.

A extensão deste trabalho para máquinas de estado probabilística é também um trabalho futuro interessante, já que aumenta a quantidade de aplicações que se beneficiariam dos resultados deste trabalho.

Uma outra extensão que pode ser bastante interessante é a geração de códigos para Controladores Lógico Programáveis (CLiP). O uso de máquinas de estado em aplicações de Controle e Automação é muito frequente, e se beneficiaria bastante da geração automática de máquinas de estado, evitando erros de programação desnecessários.

Por fim, com o intuito de tornar a máquina de estados mais completa, ações de entrada e saída poderiam ser implementadas, assim como a atribuição de prioridades entre os eventos.

Referências

AHO, A. V.; OTHERS. *Compilers: Principles, Techniques, and Tools*. 2. ed. [S.l.]: Pearson/Addison Wesley, 2007. ISBN 9780321486813.

AMOS, B. *Hands-On RTOS with Microcontrollers: Building Real-Time Embedded Systems Using FreeRTOS, STM32 MCUs, and SEGGER Debug Tools*. [S.l.]: Packt Publishing, 2020. ISBN 1838826734.

BAINOMUGISHA, E. et al. A survey on reactive programming. *ACM Computing Surveys*, v. 45, n. 4, 2013.

BEAZLEY, D. *Python Essential Reference*. 2. ed. [S.l.]: New Riders, 2001. ISBN 0735710910.

BONÉR, J.; OTHERS. *Reactive Manifesto. Reactive Manifesto, The*. 2014. Disponível em: <https://www.reactivemanifesto.org/>. Acesso em: 20 de julho de 2021.

BOYLESTAD, L. N. R. L. *Electronic Devices and Circuit Theory*. 11. ed. [S.l.]: Prentice Hall, 2012. ISBN 0132622262.

EMBARCADOS. *Sistema Embarcado – O Que É? Qual a Sua Importância?* 2023. Disponível em: <https://www.embarcados.com.br/sistema-embarcado/>.

FERREIRA, A. B. d. H. *Novo Aurélio Século XXI: O Dicionário Da Língua Portuguesa*. 3. ed. [S.l.]: Editora Positivo, 2004. ISBN 9788574724140.

FOWLER, M. *Domain-Specific Languages*. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0321712943.

FRIEDL, J. *Mastering regular expressions*. [S.l.]: O'Reilly Media, Inc., 2006.

GORELICK, I. O. M. *High Performance Python: Practical Performant Programming for Humans*. [S.l.]: O'Reilly Media, 2014. ISBN 1449361595.

GUEDES, M. *O Que É Programação Reativa? Treina WEB*. 2021. Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-programacao-reativa/>. Acesso em: 20 de julho de 2021.

GUPTA, V. et al. *P: Safe Asynchronous Event-Driven Programming*. [S.l.], 2012. Disponível em: <https://www.microsoft.com/en-us/research/publication/p-safe-asynchronous-event-driven-programming/>.

HAREL, D. *Statecharts: a Visual Formalism for Complex Systems*. 1987. Disponível em: <https://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf>. Acesso em: 15 de abril de 2022.

HERLIHY, M.; SHAVIT, N. *Art of Multiprocessor Programming, The*. [S.l.]: Morgan Kaufmann Publishers, 2008. ISBN 9780123705914.

IBM. *7090 Data Processing System*. 2023. Disponível em: <https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP7090B.html>. Acesso em: 19 de abril de 2023.

IFRAH, G. *Universal History of Computing: From the Abacus to the Quantum Computer, The*. [S.l.]: Wiley, 2000. ISBN 0471396710.

IFRAH, G. *Universal History of Computing: From the Abacus to the Quantum Computer, The*. [S.l.]: John Willey and Sons, 2005. ISBN 0471396710.

JIMÉNEZ ROGELIO PALOMERA, I. C. M. *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. [S.l.]: Springer, 2014. ISBN 9781461431428.

KUNG, P. C. C. *Domain-Specific Language (DSL) Adoption into NASA's Goddard Earth Observing System (GEOS) Model Code. NASA*. 2022. Disponível em: <https://www.nas.nasa.gov/SC22/research/project24.html>. Acesso em: 30 de dezembro de 2022.

LANGTANGEN, H. P. *A Primer on Scientific Programming with Python*. 5. ed. [S.l.]: Springer, 2016. ISBN 9783662498873.

LARK. *Grammar Reference, C2020*. 2022. Disponível em: <https://lark-parser.readthedocs.io/en/latest/grammar.html>. Acesso em: 20 de janeiro de 2022.

LEE, S. A. S. E. A. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. [S.l.]: Lulu.Com, 2011. ISBN 0557708575.

LEVELT, W. J. M. *An Introduction to the Theory of Formal Languages and Automata*. Amsterdam, the Netherlands: John Benjamins Publishing Company, 2008. 151 p. ISBN 978-90-272-3250-2.

LINGE, H. P. L. S. *Programming for Computations - Python. A Gentle Introduction to Numerical Simulations with Python 3.6*. 2. ed. [S.l.]: Springer Cham, 2019. ISBN 9783030168773.

NASA. *Design and Development of a Domain-Specific Language (DSL) Based on Hierarchical State Machines (HSM) for Model-Based Programming (MBP)*. 2020. Disponível em: <https://arcs.center/design-and-development-of-a-domain-specific-language-dsl-based-on-hierarchical-state-machines-hsm-for-model-based-programming-mbp/>. Acesso em: 20 de maio de 2024.

NOSRATI, M. Python: An appropriate language for real world programming. *World Applied Programming Journal*, v. 1, n. 2, p. 110–117, 2011.

OMG. *OMG Unified Modeling Language (OMG UML), Superstructure*. 2009. Disponível em: <https://doc.omg.org/formal/2009-02-02.pdf>. Acesso em: 26 de janeiro de 2023.

PARR, T. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. [S.l.]: Pragmatic Bookshelf, 2010. (Pragmatic Programmers). ISBN 9781934356456.

PICARD, R. *Hands-On Reactive Programming with Python: Event-Driven Development Unraveled with RxPY*. [S.l.]: Packt Publishing, 2018. ISBN 1789138728.

PYTHON. *Python Documentation, C2022. Página History and License*. 2022. Disponível em: <https://docs.python.org/3/license.html\#history-of-the-software>. Acesso em: 20 de dezembro de 2022.

RAPP, C. W. *SMC — the State Machine Compiler*. 2021. Disponível em: <https://smc.sourceforge.net/>. Acesso em: 20 de maio de 2024.

SAMEK, M. *Who Moved My State?* 2003. Disponível em: <https://drdobbs.com/who-moved-my-state/184401643>. Acesso em: 22 de abril de 2023.

SAMEK, M. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. 2. ed. [S.l.]: Newnes, 2008. ISBN 9780750687065.

SAMEK, M. *Superloop Vs Event-Driven Framework*. 2012. Disponível em: <https://embeddedgurus.com/state-space/2012/05/superloop-vs-event-driven-framework/>. Acesso em: 22 de abril de 2023.

SIPSER, M. *Introduction to the Theory of Computation*. 3rd. ed. [S.l.]: Course Technology Inc, 2013. ISBN 9780357670583.

TELLIER, D. et al. *Towards the Hierarchical State Machine Oriented Proteus Systems Programming Language*. 2020. Disponível em: <https://havelund.com/Publications/ascend-proteus-2020.pdf>. Acesso em: 20 de maio de 2024.

THURSTON, A. *Colm Programming Language*. 2021. Disponível em: <https://www.colm.net/open-source/colm/>. Acesso em: 23 de maio de 2024.

THURSTON, A. *Ragel State Machine Compiler*. 2024. Disponível em: <https://github.com/adrian-thurston/ragel>. Acesso em: 20 de maio de 2024.

TORRES, M. F. D. *Rede de Sistemas Embarcados que Utilizam Algoritmos Genéticos Aplicado na Otimização de Despacho em Mineração com Múltiplas Rotas*. Dissertação (Dissertação de mestrado) — Universidade Federal de Uberlândia, Uberlândia, 2017. Disponível em: <https://repositorio.ufu.br/handle/123456789/24300>. Acesso em: 09 de abril de 2023.

VAHID, T. D. G. F. *Embedded System Design: A Unified Hardware Software Introduction*. [S.l.]: Wiley, 2001. ISBN 0471386782.

W3C. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. 2015. Disponível em: <https://www.w3.org/TR/scxml/>. Acesso em: 25 de maio de 2024.

WIKIPEDIA. *Sistema Embarcado*. 2021. Disponível em: <https://pt.wikipedia.org/wiki/Sistema_embarcado>. Acesso em: 20 de junho de 2021.

WIKIPEDIA. *List of Common Microcontrollers*. 2023. Disponível em: <https://en.wikipedia.org/wiki/List_of_common_microcontrollers>. Acesso em: 19 de abril de 2023.

WIKIPEDIA. *Assembly Language*. 2024. Disponível em: <https://en.wikipedia.org/wiki/Assembly_language>.

YEAGER, D. P. *Object-Oriented Programming Languages and Event-Driven Programming*. [S.l.]: Mercury Learning and Information, 2014. ISBN 1936420376.