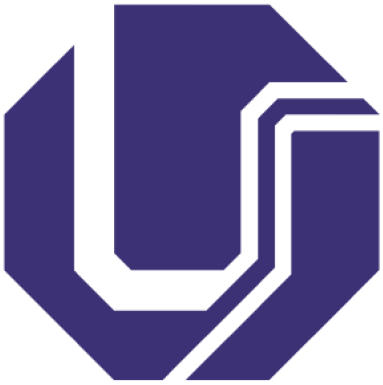
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA



FIRMWARE EM RUST: ESTRATÉGIAS DE IMPLEMENTAÇÃO E MODELOS DE PROGRAMAÇÃO CONFIÁVEIS VISANDO APLICAÇÕES CONCORRENTES EM TEMPO REAL

Cecília Carneiro e Silva

Julho
2019

CECÍLIA CARNEIRO E SILVA

FIRMWARE EM RUST: ESTRATÉGIAS DE IMPLEMENTAÇÃO E MODELOS DE PROGRAMAÇÃO CONFIÁVEIS VISANDO APLICAÇÕES CONCORRENTES EM TEMPO REAL

Dissertação apresentada ao Programa de Pós-graduação da Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia, como parte dos requisitos para a obtenção do título de Mestre em Ciências.

Banca Examinadora:

Prof. Ernane Antônio Alves Coelho, Dr. – Orientador (UFU)
Prof. Marcelo Barros de Almeida, Dr. – Coorientador (UFU)
Prof. Márcio José da Cunha, Dr. (UFU)

Prof. Rodrigo Maximiano Antunes de Almeida, Dr. (UNIFEI)

Uberlândia
2019

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

S586 Silva, Cecília Carneiro e, 1994-
2019 Firmware em Rust [recurso eletrônico] : estratégias de implementação e modelos de programação confiáveis visando aplicações concorrentes em tempo real / Cecília Carneiro e Silva. - 2019.

Orientador: Ernane Antônio Alves Coelho.
Coorientador: Marcelo Barros de Almeida.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Engenharia Elétrica.
Modo de acesso: Internet.
Disponível em: http://dx.doi.org/10.14393/ufu.di.2019.2040
Inclui bibliografia.
Inclui ilustrações.

1. Engenharia elétrica. I. Coelho, Ernane Antônio Alves, 1962-, (Orient.). II. Almeida, Marcelo Barros de , 1972-, (Coorient.). III. Universidade Federal de Uberlândia. Pós-graduação em Engenharia Elétrica. IV. Título.

CDU: 621.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

Agradecimentos

Primeiramente, agradeço a Deus pela saúde, sabedoria, fé e oportunidades que me foram concedidas.

Aos meus pais, pelo exemplo e incentivo, e pelo apoio proporcionado em todas as etapas de minha vida.

Aos Professores, pela confiança, apoio e conselhos durante todo o decorrer do mestrado.

Agradeço aos meus amigos pelo companheirismo e pelo incentivo incondicionais.

À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), pelo apoio financeiro concedido no âmbito do projeto.

Ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, pelo excelente serviço prestado.

A todos aqueles que contribuíram para a realização deste trabalho, meus sinceros agradecimentos.

Resumo

SILVA, Cecília Carneiro e. **Firmware Em Rust: Estratégias De Implementação E Modelos De Programação Confiáveis Visando Aplicações Concorrentes Em Tempo Real.** Dissertação de Mestrado – Faculdade de Engenharia Elétrica – UFU. Uberlândia, 2019.

Os sistemas embarcados são vitais na transição para um mundo mais conectado. Eles se diferem dos tradicionais sistemas de informação por uma série de fatores, largamente influenciados, pela proximidade (interseção) com o "mundo real". De forma geral, programas computacionais podem ser vistos, ou melhor, construídos sob duas perspectivas: seja a partir de uma visão próxima à máquina, ou através de abstrações de alto nível. No que tange os software embarcados, é nítido o domínio da primeira classe de atuação. As recentes falhas, todavia, propõe uma mudança de paradigma, ao acreditar que os métodos vigentes não são suficientes para lidar com as dificuldades contemporâneas. Rust, mesmo sendo uma linguagem de sistemas, se alinha com a segunda tendência de pensamento. Apoiada em verificações estáticas, e algumas validações dinâmicas, essa linguagem oferece ao desenvolvedor a possibilidade de, idiomaticamente, criar soluções confiáveis. Rust combina técnicas que, até então, eram restritas à linguagens de pesquisa, de uma forma surpreendentemente ergonômica e eficiente. O presente trabalho, amparado em modelos de computação pertinentes ao escopo, explora o uso dessa linguagem no contexto das aplicações microcontroladas. Após breve introdução, apresenta-se, como contribuição, uma biblioteca de abstração de hardware, em Rust, construída com a finalidade de traduzir propriedades "mecânicas" do hardware para abstrações seguras em software. Em seguida, são explicitados os mecanismos de controle da complexidade proveniente da natureza física dessas aplicações. Um modelo de temporalização via *timer-wheels* e dois padrões de construção de software, de alto nível, complementares o modelo de baixo nível (tarefas e recursos) pregado pela *framework* RTFM (*Real-Time for The Masses*) são as demais contribuições deste trabalho. Em suma, as técnicas aqui apresentadas foram pensadas para adaptar a linguagem Rust às características das aplicações embarcadas. O resultado são software mais confiáveis e seguros.

Palavras-chave: *Cyber Physical Systems*, Engenharia de Software, Linguagem Rust.

Abstract

SILVA, Cecília Carneiro e. **Firmware In Rust: Reliable Implementation Strategies And Programming Models Targeting Concurrent And Real-Time Applications.** Masters Dissertation – Faculty of Electrical Engineering – UFU. Uberlândia, 2019.

Embedded systems are particularly vital in the transition to a more connected world. They differ from traditional information systems by a series of factors, largely influenced by proximity (intersection) with the "real world". In general, computer programs can be seen, or rather, constructed from two perspectives: either from a computer-oriented view, or through computation-oriented abstractions. With regard to embedded software, the domain of the first class of thinking is clear. The recent flaws, however, propose a paradigm shift, believing that current methods are not enough to deal with contemporary difficulties. Rust, even though it is a systems language, aligns with the second thinking trend. Based on static verifications, and some dynamic validations, this language gives the developer the possibility to create trustworthy solutions. Rust combines techniques that until then were restricted to search languages in a surprisingly ergonomic and efficient way. The present work, based on computation models pertinent to the scope, explore the use of this language in the context of microcontrolled applications. After a brief introduction, the contribution is presented, a hardware abstraction library, in Rust, built for the purpose of translating "mechanical" properties of the hardware into secure abstractions into software. Then, mechanisms to control the complexity coming from the physical nature of these applications are explained. A “timer-wheel” timing model and two high-level software building standards, complementary to the low-level model (tasks and resources) preached by the RTFM framework (Real-Time for The Masses) are the other contributions of this work. In short, the techniques presented here were designed to adapt the Rust language to the characteristics of embedded applications. The result is more reliable and secure software.

Keywords: *Cyber Physical Systems*, Rust Language, Software Engineering.

Lista de Figuras

Lista de Abreviaturas

ABI	Application Binary Interface (Interface Binária de Aplicação)
API	Application Programming Interface (Interface de Programação de Aplicação)
CMSIS	Cortex Microcontroller Software Interface Standard
CPS	Cyber Physical Systems
CPU	Central Processing Unit
CQRS	Command Query Responsibility Segregation
DMA	Direct Memory Access
DSL	Domain Specific Language (Linguagem de Domínio Específico)
FIFO	First In First Out
FSM	Finite State Machine (Maquina de estados finita)
GPL	General Purpose Language
HAL	Hardware Abstraction Layer
IoT	Internet of Things (Internet das coisas)
LTO	Link Time Optimizations
MCU	Microcontrolador

MIT	Massachusetts Institute of Technology
MoC	Models of Computation
MSP	Main Stack Pointer
NRZ	Non Return-to Zero
NSF	National Science Foundation
NVIC	Nested Vectored Interrupt Controller
OO	Orientação a Objeto
OOL	Object Oriented Language (Linguagem Orientada a Objetos)
PC	Program Counter
RFC	Request For Comments
RTFM	Real Time For the Masses
SM	State Machine (Maquina de estados)
SRP	Stack Policy Resource
SVD	System View Description
TQ	Timer Queue
TW	Timer Wheel
uP	Microprocessador
USART	Universal Synchronous Asynchronous Receiver-Transmitter

Sumário

Capítulo 1

Introdução

No imaginário popular, figura a ideia que computadores são apenas máquinas de mesa criadas para processar informações. Essa visão, entretanto, não é fidedigna com a realidade. Atualmente, a maioria dos dispositivos computacionais em uso são menos visíveis e possuem funções bem mais específicas. São conhecidos como sistemas embarcados. Esses computadores são encontrados em diversos equipamentos e nem sempre são visíveis ao usuário final. Eles controlam os freios de um carro, digitalmente codificam sinais entre celulares e estações, comandam robôs em fábricas, controlam aviões, trens e redes de energia inteligentes (*smart grid*).

1.1 Contextualização e Motivação

Computadores são embarcados em aplicações desde o início da computação, *Whirlwind*, por exemplo, criado no fim da década de 1940, no Instituto de Tecnologia de Massachusetts (MIT), foi originalmente concebido para controlar, em tempo real, um simulador de voo (WOLF, 2012). Por mais que não seja um tópico recente, os embarcados foram negligenciados por muito tempo pela comunidade acadêmica (LEE, Edward Ashford; SESHIA, 2012). Percebeu-se, há pouco tempo, que as técnicas de engenharia requeridas para criá-los e analisá-los se diferem das tradicionalmente aplicadas em sistemas de informação.

Acreditava-se que as dificuldades em desenvolver tais sistemas estavam relacionadas, apenas, aos seus recursos limitados: baixo poder de processamento, fonte de energia limitada (bateria) e memória reduzida (LEE, Edward Ashford; SESHIA, 2012). Restringindo as técnicas de engenharia

à otimizações de design. Recentemente, entendeu-se que o principal desafio dos sistemas embarcados advêm da sua interação com o mundo físico (LEE, Edward Ashford; SESHIA, 2012). Levando a *National Science Foundation* (NSF), em 2006, a criar o termo *cyber-physical systems* (CPS) (LEE, Edward A, 2008), referência à intercessão entre computadores e processos físicos. Eles são a convergência entre informação, computação, comunicação e controle.

O impacto econômico das aplicações CPS é inegável e como possuem papel vital na Indústria 4.0, tornaram-se política de estado em diversos países (LEE, J.; BAGHERI; KAO, 2015). Embora decorra do rápido desenvolvimento dos dispositivos *cyber-physical* durante a última década, cabe ressaltar que a quarta revolução industrial traz uma maneira diferente de pensar no sistema como um todo (HE; JIN; KARLTUN, 2016; LEZOCHE; PANETTO, 2018). Uma classe especial de dispositivos físico-computacionais são os equipamentos IoT (*Internet of Things*) (LINDNER, A., 2015). Eles apresentam como diferencial a interconexão através de uma rede, como a Internet (MATTERN; FLOERKEMEIER, 2010).

A esse trabalho interessa, prioritariamente, o componente lógico desses dispositivos, o software. Enquanto a ciência da computação se esforça sistematicamente para abstrair o "mundo físico", os softwares embarcados devem interagir com ele. Tempo, robustez, reatividade, concorrência são requisitos não funcionais em aplicações tradicionais; para os sistemas embarcados são requisitos funcionais (LEE, Edward A, 2002). Nas raízes da computação está a noção de que software é a realização de procedimentos matemáticos, mapeando entradas em saídas. Programas embarcados, no entanto, não têm por objetivo transformar dados e sim interagir com o mundo físico (HALBWACHS, 1993; LEE, Edward A, 2002). A confiabilidade é particularmente importante a esses sistemas, para muitos é a funcionalidade mais importante (HALBWACHS, 1993). As consequências de uma falha podem envolver vidas humanas e resultar em volumosas perdas financeiras.

Com operações simples e forte controle sobre o hardware subjacente, a linguagem C foi criada para contornar os obstáculos enfrentadas no desenvolvimento de sistemas operacionais na década de 1970, principalmente a interoperabilidade entre máquinas. Sua proximidade com o baixo nível, acompanhada das abstrações de baixo custo, a fizeram popular também no mundo físico-computacional. Contudo, o crescimento acelerado e o aumento da complexidade sugerem novas óticas de atuação, priorizando métodos de construção mais propícios para lidar com as

dificuldades contemporâneas (WOLF, 2016). Para esse software maiores, em tamanho e funcionalidades, é fundamental ter a habilidade de esconder, seguramente, detalhes ao longo do processo de desenvolvimento.

1.2 Objetivos e contribuições

O presente trabalho atua no tema propondo o uso da linguagem de programação Rust auxiliada por modelos de computação (*Models of Computation*, MoC) pertinentes ao contexto dos sistemas físico-computacionais. Nesse sentido, são apresentados métodos e abstrações que visam viabilizar a construção de aplicações mais confiáveis. Fundamentadas em técnicas formais, verificações estáticas e validações dinâmicas, as soluções apresentadas trazem consigo conceitos computacionais nobres; alguns raramente empregados no domínio em questão. Sem perder de vista a eficiência das soluções, esse trabalho busca ressaltar que performance e confiabilidade não precisam ser atributos excludentes.

Para chegar ao objetivo final, algumas metas específicas precisam ser atendidas. No escopo desse trabalho está uma breve introdução à linguagem Rust, chegando à sua aplicação em microcontroladores. Desse ponto resulta a primeira grande contribuição deste texto: a implementação de uma biblioteca de abstração de hardware para o microcontrolador adotado. Há um cuidado especial em elencar os passos e conceitos que nortearam a criação da mesma e, como ela se difere das abstrações tradicionalmente usadas para os mesmos fins em outras linguagens de programação.

O restante do texto se preocupa em conciliar as características físicas, as quais esses sistemas estão sujeitos, com modelos de computação apropriados para lidar com tal complexidade. No primeiro capítulo tocante a esse assunto, a plataforma RTFM é apresentada, elucidando como as "tarefas" e os "recursos" ajudam a controlar os efeitos da concorrência nos sistemas embarcados. Em particular, como possibilitam o compartilhamento seguro de dados entre os manipuladores de interrupção e o fluxo principal. Adiante, têm-se a próxima contribuição: uma "roda" de temporização via software, escrita em Rust, adaptada às limitações do dispositivo aqui empregado.

O design de baixo nível, praticado pelo RTFM (*Real-Time for The Masses*), traz eficiência às soluções, entretanto, incorre em problemas característicos à construções próximas ao hardware. Após enumerar essas dificuldades, o presente trabalho, propõe, como contribuição, dois modelos

de extensão ao RTFM. Eles não são invasivos, nem requerem mudanças internas à plataforma.

O primeiro, é a interpretação de uma arquitetura norteada por eventos, em particular, uma fila de reação de eventos. Ao promover a inversão do fluxo de controle, esse modelo clarifica a ligação entre as tarefas, facilitando a concepção da aplicação e futuras manutenções. A segunda proposta é um sistema de descrição (criação) de máquinas estados, seguro e automatizado. Estados e transições, ao separar as preocupações, facilitam a compreensão da dinâmica da aplicação. Para ambos os casos, o presente texto aborda suas respectivas implementações, escolhas de design e uma aplicação exemplo.

Todos os software desenvolvidos em favor deste texto estão disponíveis em repositórios abertos referenciados no decorrer da dissertação. Os trechos de códigos, mostrados ao longo deste texto, foram didaticamente posicionados de forma a fortalecer as explicações textuais e podem não representar, em sua totalidade, as implementações criadas. Ao final, espera-se que o leitor tenha ciência dos obstáculos que conduziram a concepção desse trabalho e, consequentemente, como as abstrações propostas aumentam a confiabilidade das aplicações resultantes.

1.3 Organização da Dissertação

Esta dissertação está organizada em 7 capítulos, descritos a seguir.

O primeiro capítulo tem por objetivo delimitar e apresentar os objetos de estudo deste trabalho, bem como justificar e dar relevância às questões pesquisadas. Ao fim, ele ambienta o leitor à estrutura que será observada nos capítulos seguintes.

Nos Capítulos 2 e 3 são expostos os principais fundamentos teóricos associados ao trabalho, além de justificá-los como objeto de estudo dessa dissertação. Engenharia de software, sistemas reativos, o núcleo de escalonamento RTFM e linguagem Rust são apresentados como os quatro pilares de sustentação desse trabalho. Os fundamentos apresentados nesse capítulo justificam as afirmações e escolhas feitas no restante do texto.

No quarto Capítulo discorre-se sobre os aspectos (e tópicos) relevantes a uma aplicação Rust para microcontroladores. Nele, é demonstrado a evolução de uma abstração ingênua para uma implementação robusta e confiável, passando por conceitos e ferramentas pertinentes ao tema. Aqui, incrementalmente, será mostrado, ao leitor, como as abstrações Rust possibilitam a cons-

trução de aplicações embarcadas mais seguras, resultando na biblioteca de abstração de hardware implementada como contribuição. Ao fim, é demonstrado a viabilidade das soluções propostas.

No Capítulo 5 é apresentado a adaptação a linguagem Rust, através da *framework* RTFM, para trabalhar em aplicações embarcadas concorrentes. Serão abordadas as concepções relativas à criação de uma aplicação de "tempo real" em Rust, pontuando ferramentas auxiliares e estruturas de dados aptas a executar sob tais condições. Revela-se também, um hiato ao tentar usar essa plataforma no microcontrolador escolhido por esse trabalho, justificando a criação a biblioteca de rodas de *timers*, como contribuições válida.

No sexto Capítulo discorre-se sobre as dificuldades encontradas ao criar aplicações maiores (e mais complexas), a partir das primitivas apresentados até aqui. Após contextualizar o porquê dos padrões de arquitetura e construção propostos como contribuição, há a descrição das soluções implementadas. Os fundamentos teóricos e ferramentas auxiliares conduzem a explicação dos modelos proposto. Nas seções internas pertinentes, uma pequena aplicação exemplifica como tal modelo ajuda a abstrair problemas característicos de uma aplicação RTFM.

Por fim, o Capítulo 7 realiza o fechamento da dissertação, conclui o que foi apresentado, reafirmando as principais contribuições que este trabalho pode trazer à comunidade científica e à sociedade. Termina pontuado possíveis pontos de continuação e trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

Nesse capítulo, são apresentados conceitos e definições de tecnologias relacionadas ao presente trabalho. Em particular, a engenharia de software, os sistemas embarcados e os sistemas tempo real serão abordados.

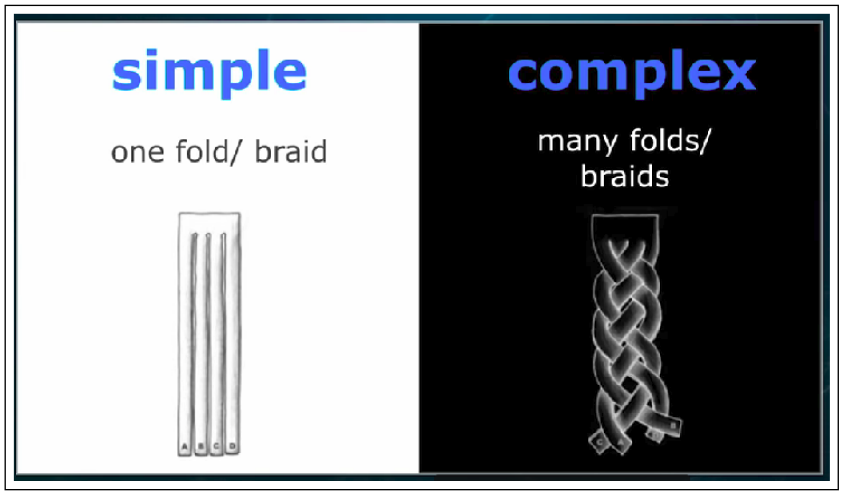
2.1 Engenharia de Software

O termo complexidade vem do latim 'plex' (dobras) e significa com dobras ou traças, Figura 2.1. Software complexo é aquele em que a interligação dos componentes não é clara, formando uma trama difícil de ser compreendida. Segundo (BROOKS, 1986), as dificuldades podem ter origem essencial ou acidental. Enquanto a complexidade essencial tem suas raízes no problema que se deseja solucionar, a complexidade acidental deriva-se da implementação do sistema. Fruto de confusões, esse conceito não tem ligação direta com a complexidade ciclomática de um programa, embora possuam a palavra em comum.

Entre as atribuições da engenharia de software está a garantir a qualidade das soluções geradas; portanto, reduzir e controlar os acidentes. Entender a solução é pre-requisito para evitar problemas, dando a complexidade o título: maior causa de problemas do desenvolvimento de software (MOSELEY; MARKS, 2006; NGUYEN-DUC, 2017). Os pejorativamente conhecidos como "códigos espaguete" geram diversas consequências negativas, principalmente nos custos manutenção (NGUYEN-DUC, 2017).

Abstração é a habilidade de filtrar informações selecionando apenas os aspectos relevantes

Figura 2.1: Simples, ou sem tranças. Complex, ou com tranças.



Fonte: (NASH, 2018)

ao contexto em análise. Esconder os comportamentos irrelevantes permite exaltar as características importantes. Componentização, modularização (PARNAS, 1972), convencionamento de interfaces e metalinguagem são abstrações empregadas em linguagens de programação para limitar a complexidade acidental. Desde 1968, com a primeira menção da crise do software (NAUR; RANDELL, 1968), estudiosos buscam formas de tornar mais ergonômico o desenvolvimento de aplicações computacionais. Eles se apoiam na comprovada ineficiência humana em armazenar muitas informações simultâneas (MILLER, 1955).

2.1.1 Padrões de Construção de Software

Os padrões (*patterns*) capturam experiências comprovadamente funcionais, ajudando a promover boas práticas de projeto (BUSCHMANN et al., 1996). Eles permitem que todos aproveitem dos conhecimentos desenvolvidos por especialistas. Dificilmente enfrenta-se um problema completamente original, práticas já validadas ajudam a construir um bom software. Padrões controlam o desconhecido, indicam componentes necessários, detalhes que devem ser escondidos e as características a serem exaltadas. São usados como documentação de alto nível e fornecem vocabulário comum a todos os envolvidos no projeto (BUSCHMANN et al., 1996).

Um padrão de arquitetura clarifica a estrutura do programa e a divisão em subsistemas, especificando suas responsabilidades e interfaces de comunicação. Padrões de design estão um degrau

abaixo, cuidam dos subsistemas, refinando-os em módulos menores e componentes.

Os idiomas são o nível mais baixo de padronização, preocupam-se em como se expressar em determinada linguagem de programação. Deixando claro que técnicas empregadas em um dialeto nem sempre são ideais para os outros. Uma codificação idiomática indica a forma mais concisa, conveniente e comum de realizar uma tarefa nesse idioma (MANSFIELD, 2016).

Em (NORVIG, 1996), Peter Norvig argumenta que as estratégias de construção (padrões de design) devem ser vista como provérbios e não como *templates*. Linguagens poderosas incorporam, ao máximo, as abstrações usuais, evitando que essas sejam descrições que somente os programadores experientes sabem.

2.1.2 Dependabilidade

A lei de Conway (*Conway's law),* (CONWAY, 1968)*,* prega que os sistemas computacionais refletem os valores organizacionais de seus criadores. Enquanto no passado se tinha apenas uma pessoa trabalhando sozinha, o foco atual está a colaboração e divisão de tarefas (PRESSMAN, 2000). Os métodos de gestão de projetos evoluíram para se adequar aos novos valores. É necessário que o ecossistema de desenvolvimento, linguagens e ferramentas, também de adequem ao dinamismo dos novos tempos.

O período de desenvolvimento é cada vez menor e a habilidade de inovar com rapidez é determinante para o sucesso (LASI HEINER et al., 2014). Enquanto no século passado, o tempo gasto para transformar descobertas tecnológicas em produtos comerciais era de 30 anos, hoje é necessário chegar ao mercado em dois ou três anos, estando, provavelmente, obsoleto em 5 anos (LEVESON, 2011). A rápida evolução nos tira a habilidade de "aprender com o erro" (LEVESON, 2011), gerando graves consequências à segurança dos sistemas. As recentes falhas envolvendo os software embarcados levam a acreditar que estamos enfrentando uma crise existencial (WOLF, 2016).

Os fatos acima ratificam a busca por ferramentas e arquiteturas inerentemente mais seguras. Verificação formais passaram a ocupar um lugar de destaque, (LEE, Edward Ashford; SESHIA, 2012; LAMPORT, 2008; FARIA, 2008; VERHULST et al., 2011). Elas permitem válidações, como testes e simulações, para provar que os sistemas modelados possuem certas propriedades desejadas. Técnicas que no passado eram descartadas por demandar alto poder computacional, hoje,

podem fazer parte do processo de desenvolvimento. Construir corretamente é, além de tudo, mais barato que corrigir depois (AMEY, 2002).

A dependabilidade de um sistema computacional é sua habilidade em entregar um serviço justificadamente confiável (AVIZ; LAPRIE; RANDELL, 2000). Um software seguro engloba uma série de características que o tornam menos vulneráveis a falhas. Disponibilidade, confiabilidade, segurança, integridade e facilidade de manutenção norteiam esse conceito interativo (AVIZ; LAPRIE; RANDELL, 2000).

A linguagem Rust, usando métodos formais leves como um poderoso sistema de tipos, tenta ao máximo garantir a confiabilidade dos códigos produzidos. O Capítulo 3 apresenta essa linguagem e suas principais características, mostrando como essas se refletem nas aplicações criadas. Com o uso extensivo de mecanismos de verificação estáticos, Rust, enaltece valores como integridade e robustez. Ao longo do texto são mostradas (e propostas) alternativas para, aproveitando-se da composibilidade da linguagem Rust, transformar problemas de execução em falhas durante a compilação.

2.2 Sistemas Embarcados

Sistemas reativos são sistemas computacionais que reagem continuamente com o ambiente no qual estão inseridos, a taxas definidas pelo ambiente (HALBWACHS, 1993). Esse termo foi introduzido em (BERRY, 1989; HAREL; PNUELI, 1985) para contrastá-los aos sistemas transformacionais - entradas disponíveis no início da execução produzem saídas ao final da execução - e aos sistemas interativos - continuamente interagem com o ambiente porém a taxas definidas pelo sistema. Nota-se que as aplicações *cyber-physical* são, em sua maioria, sistemas reativos.

Os software embarcados executam em máquinas que não são, acima de tudo, computadores; são aviões, carros, telefones, brinquedos, monitores de segurança, entre outros (LEE, Edward A, 2002). Com isso, eles adquirem propriedades do mundo com o qual interagem: levam tempo, consomem energia e não terminam (a menos que falhem) (LEE, Edward A, 2002). Das definições acima deriva-se duas características fundamentais aos CPS: concorrência e rigorosos requisitos de tempo. Essas restrições devem ser expressas nas especificações do sistema, consideradas durante o projeto e verificadas na implementação (HALBWACHS, 1993).

A ciência da computação tendeu a negligenciar os sistemas embarcados por considerar sua "fisicalidade" confusa. O design dessas aplicações pouco se beneficiou dos recursos de abstração do século XX (LEE, Edward A, 2002). E raramente os programadores de sistemas embarcados possuem formação em ciência da computação, tentem a ser especialistas no domínio da aplicação, o que é provavelmente o apropriado. Todavia, a crescente complexidade (e tamanho) das soluções acende um alerta para a necessidade de adaptação às dificuldades atuais. Eles são mais do que "pequenos computadores" e precisam de técnicas que enalteçam suas características físicas e forneçam a segurança necessária (LEE, Edward A, 2002).

2.2.1 Tempo real e concorrência

A distinção entre sistema de tempo real e os outros se baseia no fato de que para a primeira classe a resposta no tempo é mais do que importante, é crucial (HALBWACHS, 1993). Sua exatidão não depende somente do resultado lógico da computação, importa também o tempo em que ele foi produzido. E, como raramente interagem com apenas um processo físico, as aplicações embarcadas são naturalmente concorrentes. Elas podem reagir, simultaneamente, aos estímulos de uma rede de comunicação e a uma variedade de sensores, além de oportunamente controlar os atuadores.

O comportamento dos sistemas de tempo real ao não cumprir os requisitos de tempo, os separam em 3 categorias (BURNS; WELLINGS, 2009). Para *hard real-time systems* é absolutamente mandatório que as respostas ocorram dentro do limite especificado. *Soft real-time systems* são aqueles para os quais a resposta no tempo é importante, mas o sistema continua funcionando se os limites não são atendidos ocasionalmente. *Firm real-time systems* possuem a capacidade de lidar com perdas de prazo ignorando respostas após o limite determinado.

As aplicações embarcadas estão sujeitas a uma série de requisitos contraditórios (LEE, Edward A, 2016): adaptabilidade x repetibilidade, alta performance x baixo consumo, assincronismo x cooperação. A existência de todos esses obstáculos torna o tema relevante para o mundo acadêmico. No que tange a criação de software, as soluções são, normalmente, categorizadas segundo suas abstrações. Cada qual traz consigo uma série de características.

2.2.2 Estratégias de desenvolvimento

Soluções em *bare metal,* usando linguagens de programação de propósito geral, como C, concedem ao programador todo o controle sobre o hardware. Com o preço de transmitir também toda a responsabilidade sobre a correta execução da aplicação. Para sistemas simples essa alternativa é viável, entretanto, em casos mais complexos tem-se dificuldades de escalabilidade e reuso de código (LINDNER, A., 2015).

Pode-se deslocar para um sistema operacional de tempo real (RTOS) a responsabilidade de "absorver" o hardware, levando para ele o escalonamento do sistema (LINDNER, A., 2015). Essa alternativa possibilita o reuso de código porém aumenta a complexidade da aplicação e possui custos de execução. A maioria dos RTOSs baseia seu modelo de concorrência no uso de *threads*, processos com memória compartilhada. Preocupações a respeito da confiabilidade e predictibilidade dos *threads* foram expressas em (LEE, Edward A, 2006). Nesses sistemas, é obrigação do programador coordenar o gerenciamento dos recursos, enquanto implementa as propriedades de tempo real. São exemplos de RTOS: FreeRTOS (AMAZON WEB SERVICES, 2019) e Nucleus (MENTOR, 2019).

Linguagens paralelas de propósito geral como Ada (ADA WORKING GROUP, 2001) fornecem primitivas de alto nível para estruturar programas e dados. Esses mecanismos de comunicação e sincronização, como *rendezvous*[1], são mais claros do que memória compartilhada. No nível mais baixo eles se apoiam no mesmo esquema assíncrono de execução (HALBWACHS, 1993), mas para a aplicação são mais seguros e previsíveis. A ampla adoção do modelo baseado em *threads* pelas demais linguagens de programação, tornam os artifícios aqui prescritos, conceitos fora do dicionário da maioria dos programadores.

Na abordagem síncrona a noção de tempo é incorporada ao design de uma forma diferente. Como nos circuitos síncronos, as primitivas são considerados instantâneos (HALBWACHS, 1993). A noção física do tempo é substituída pelo conceito de ordem entre eventos: simultaneidade e precedência (HALBWACHS, 1993). Na prática, essa hipótese se baseia na suposição de que o programa é capaz de reagir rápido o suficiente para capturar as entradas de forma adequada.

As linguagem síncronas (reativas) são uma tentativa de conciliar concorrência e determinismo,

[1]Mecanismo baseado em um modelo de interação cliente/servidor. Uma "tarefa servidor" declara o conjunto de serviços que está preparada para oferecer às demais tarefas (através de "entradas" públicas). O *rendezvous* controla a chamada e aceitação de um "serviço", assegurando o correto sincronismo do sistema.

são exemplos: Esterel (BERRY, Gérard, 2000), Lustre (HALBWACHS, 1991), Céu (SANTANNA; IERUSALIMSCHY; RODRIGUEZ, 2015). O projeto *Precision Timed (PRET) Machines* (CHESS, 2015; LEE, Edward A, 2016), foi além e mostrou como estender tais conceitos para o conjunto de instruções - ISA (*Instruction-Set Architectures*). Os métodos e formalismos encontrados nessa vertente são impressionantes mas disruptivos, dificultando sua popularização.

Com a premissa de maximizar as vantagens e permitir análises estáticas sem pagar um custo elevado, estão as *frameworks* concorrentes e de tempo real. Quando absorvem a tarefa de escalonar o sistema, elas são conhecidas como sistemas operacionais leves. Essas ferramentas se fundamentam em modelos de computação específicos, como sistemas de atores, máquinas de estado, redes *Petri* e/ou protocolos de compartilhamento de recursos. São criadas para estender a capacidade das linguagens assíncronas, incorporando-lhes características intrínseca ao domínio em questão. Normalmente, não possuem a generalidade das abordagens citadas acima, o que nem sempre é uma desvantagem.

Aqui, a *framework* RTFM, Seção 2.3, irá aprimorar o modelo de concorrência da linguagem Rust adaptando-o à natureza das aplicações microcontroladas. O protocolo de compartilhamento de recursos e o escalonador simples ajudarão a lidar com o concorrência proveniente do ambiente. Apoiados nos mecanismos de interrupção em hardware, eles irão absorver boa parte da complexidade envolvida. Um sistema de construção de máquinas de estados e uma arquitetura norteada por eventos completam as abstrações propostas.

2.3 Sistemas em tempo real e RTFM

Em RTOSs baseados em *threads*, o gerenciamento incorreto de recursos pode levar a *data-races* e *deadlock*s (LINDNER, A., 2015). Após a compilação, pouco apoio é dado ao programador em termos de validação do programa, requerendo testes manuais. No modelo de computação proposto pelo RTFM, os processos são tratados de forma não uniforme e atenção especial é dada ao manuseio de recursos compartilhados (LINDNER, A., 2015), responsáveis por boa parte da complexidade dos sistemas reativos.

RTFM é um conjunto de ferramentas de programação reativa, baseada em eventos e recursos (LINDGREN; LINDNER, M. et al., 2015; LINDNER, M.; LINDNER, A.; LINDGREN, 2016). Seu prin-

cipal objetivo é facilitar a criação de aplicações de tempo real, confiáveis e eficientes, amparadas por verificações estáticas. O *framework* é resultado dos esforços do grupo de pesquisa em sistemas embarcados da universidade de tecnológica Luleå *(Luleå University of Technology (LULEÃ, 2019)).* Surgiu, desde sempre, com o propósito de popularizar soluções de tempo real para todos (*Real-Time for The Masses*).

Seu ensino é usado com introdução a Linguagens de Domínio Específico, *Domain Specific Language* (DSL), na matéria de construção de compiladores para o curso de mestrado em ciência da computação (LINDNER, M.; LINDNER, A.; LINDGREN, 2015). RTFM-*lang* foi originalmente implementado em OCaml/Menhir, suportanto primitivas em C. Dos estudos preliminares, interessa, ao presente trabalho, o *RTFM-Kernel*, núcleo minimalista de controle, baseado no modelo de computação proposto pelo *Stack Resources Policy* (SRP) (BAKER, 1990), executado diretamente no mecanismo de controle de interrupção.

2.3.1 Algoritmos de escalonamento

Tarefas em sistemas de tempo real podem ser separadas em 3 grupos: periódicas, aperiódicas e esporádicas. Tarefas com intervalo de estímulo regular são conhecidas como periódicas. Elas são comumente usadas para processar dados de sensores ou verificar estados em intervalos regulares. Quando não há um padrão de estímulo, a tarefa é dita aperiódica. Tarefas aperiódicas são usadas para manipular eventos de natureza aleatória e, convencionalmente possuem *soft deadlines*. Quando o limite de execução de uma tarefa não periódica precisa obrigatoriamente ser atendido ela é nomeada esporádica (SPRUNT; SHA; LEHOCZKY, 1989).

Algoritmos de escalonamento desempenham papel vital nas aplicações, determinam qual tarefa será despachada para execução no momento em que várias estão disponíveis. Nos sistemas de tempo real, o objetivo principal de um escalonador é garantir que os requisitos de tempo serão cumpridos. Os algoritmos podem ser *offline* (estáticos) ou *online* (dinâmicos).

Quando *offline,* o escalonador é rígido e calculado a priori. O caráter estacionário permite ao projetista predizer o comportamento do sistema, porém essa rigidez inviabiliza sua utilização em boa parte dos casos.

Nos escalonadores *online* a decisão de qual processo será executado é decidida em tempo de execução, tornando-os susceptíveis a trabalhar com tarefas não periódicas. O critério de troca de

contexto pode ser temporal ou norteado por eventos. Em algoritmos temporais, a cada intervalo de tempo definido, uma nova tarefa é escalonada para execução. Caso seja norteado por eventos, o escalonador pode ter seu contexto modificado a cada evento, como a chegada ou finalização de uma tarefa.

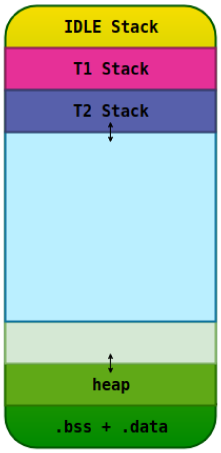
Escalonadores podem ser preemptivos ou não. Quando todas as tarefas possuem o mesmo nível de urgência e não devem influenciar na execução da outra, o algoritmo não deve ser preemptivo. Como predominantemente os sistemas são compostos por processos com diferentes níveis de necessidades, suas tarefas são priorizadas. *Rate Monotonic Algorithm (RMA)*, *Earlier Deadline First (EDF)* e *Deadline Monotonic Algorithm (DMA)* são os três exemplares mais conhecidos dessa categoria de escalonadores. A tarefa de maior prioridade pronta para executar será a escolhida para atuar.

Enquanto nos algoritmos RMA e DMA o nível de prioridade é atribuído em tempo de compilação; no EDF a prioridade é variável durante a execução. No RMA, as tarefas são priorizadas levando em consideração sua periodicidade; quando menor o tempo entre estímulos maior a prioridade. Como o nome indica, para o algoritmo DMA, a propriedade determinante é o tempo limite de execução; quanto menor a *deadline* mais rápido a tarefa precisa ser atendida e, portanto, maior sua prioridade.

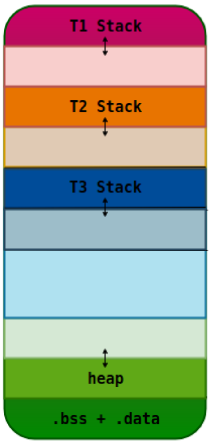
No EDF, a cada evento oriundo no sistema, o tempo até a completa execução é calculado para todos os processos, quanto menor o tempo limite, maior a prioridade atribuída. Por mais que seja um “algoritmo ideal” - caso exista, o algoritmo irá encontrar a solução de escalonamento - para sistema uniprocessados, as soluções apoiadas no EDF normalmente não são aplicadas comercialmente. O comportamento pouco determinístico e as reações em cadeia inesperadas dificultam a análise dos programas.

Figura 2.2: *Stack* de execução.

(a) Todas as tarefas compartilham a mesma pilha.



(b) Em um sistema com *threads*, cada possui sua pilha de execução.



Fonte: Autoria própria.

Em determinados casos, os projetistas desejam um escalonador que possibilite um controle mais rebuscado do fluxo de execução. Ele deve permitir, por exemplo, que uma tarefa pause sua execução (`yield`) enquanto espera alguma condição; devolvendo o contexto ao escalonador. Para tal, o sistema precisa ter aptidão de retornar à tarefa no estado em que ela estava ao pausar. Salvar o estado atual e retornar a esse ponto quando desejado é conhecido como continuação (*continuation*).

Parte dos RTOSs tratam esse problema separando a pilha de execução, cada ganha sua pilha própria. O programador é responsável por delimitar a área de cada *stack* e o *kernel* as usará para operar a troca de contexto, Figura 2.2b. Caso o tamanho atribuído não seja suficiente, problemas como o transbordamento de pilha podem ocorrer. Outros sistemas, como Romantiki OS (GLISTVAIN; ABOELAZE, 2010), usam a "continuação local", e emulam esse funcionamento com variáveis globais e corotinas, em uma única pilha de execução.

No escalonador do RTFM, os bloqueios de execução assíncronos citados acima não estão disponíveis, apenas as trocas preemptivas (síncronas) são permitidas. As tarefas são finitas e diretamente ligadas a um manipulador de interrupção em hardware. Assim, devem executar até o fim. Como a pilha de execução é única, fig. 2.2a, para voltar a execução de `T1` é necessário finalizar

T2 antes. A menor expressividade é compensada pela eficiência e simplicidade da solução. A *framework* pode se apoiar, completamente, no mecanismo de tratamento de interrupção em hardware. O escalonador software precisa apenas atar a tarefa ao seu manipuladores corresponde, o restante do processo é transparente à aplicação.

Recursos são qualquer estrutura de software que pode ser usada por um processo para avançar sua execução (BUTTAZZO, 2011). Normalmente são estruturas de dados, conjunto de variáveis ou registros de um periférico. Quando particular a apenas um processo, o recurso é conhecido como privado; quando compartilhado entre várias tarefas, chama-se recurso compartilhado. A proteção de um recurso compartilhado contra acessos concorrentes é conhecida como acesso exclusivo.

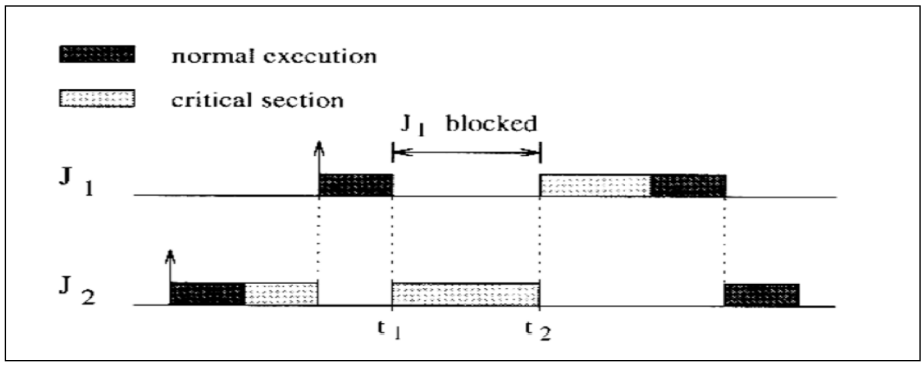
Corridas de dados e inconsistências em recursos compartilhados são evitadas através de uma política de exclusão mútua. O código executado sob tais condições é denominado seção crítica. Para adquirir um recurso compartilhado é necessário que esse esteja livre, ou seja, não esteja sob uso de outra tarefa. Quando isso não é possível é dito que o processo está bloqueado por tal recurso. Dentre os mecanismos de sincronização destacam-se mutex e semáforos, cada um dos métodos possui particularidades e problemas associados. Ao escopo deste trabalho interessa as consequências relacionadas à execução de mutexes em uniprocessadores.

Inversão de prioridade ocorre quando uma tarefa de menor prioridade bloqueia a execução de outras mais prioritárias. O evento de bloqueio pode ser de duas naturezas: limitado e ilimitado. A inversão de prioridade limitada ocorre quando uma tarefa de menor prioridade reivindica um recurso bloqueando tarefas mais prioritárias que necessitam do mesmo recurso, Figura 2.3. Esse processo é natural e quando usado de forma racional não traz graves consequências.

Já a inversão de prioridades ilimitada acontece quando tarefas intermediárias estendem a duração desse processo natural, possivelmente para sempre. Na Figura 2.4 é exemplificado a inversão de prioridades ilimitada, evidenciando graficamente como a predictibilidade e requisitos temporais ficam comprometidos com ocorrência desse evento. No tempo `t3`, J3 requer acesso exclusivo a um recurso, impedindo que J1 (mais prioritária) seja executada. Em `t4`, J2, mais prioritária do que J3 e menos prioritária do que J1, entra em estado de prontidão. Como essa tarefa não depende do recurso requerido por J3 e tem maior prioridade, ela tem sua execução escalonada aumentando o tempo de bloqueio da tarefa J1.

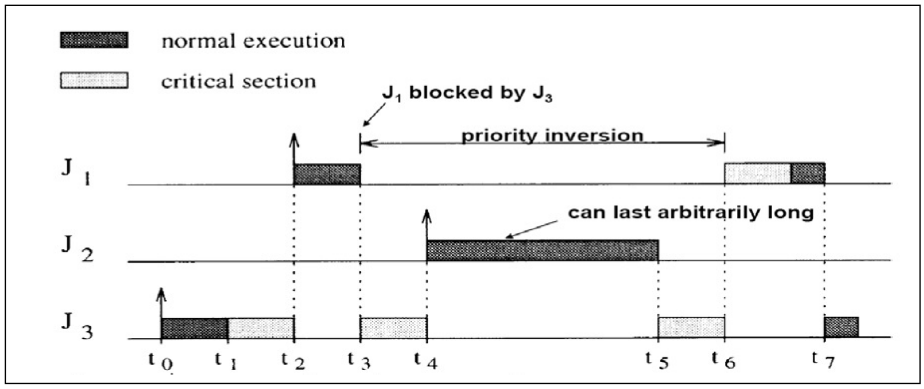
Um caso conhecido no qual a execução de tarefas intermediárias atrapalhou o comportamento

Figura 2.3: Inversão de prioridade limitada. Em t1, J2 tarefa menos prioritária inicializa uma seção crítica bloqueando a execução de J1 (mais prioritária).



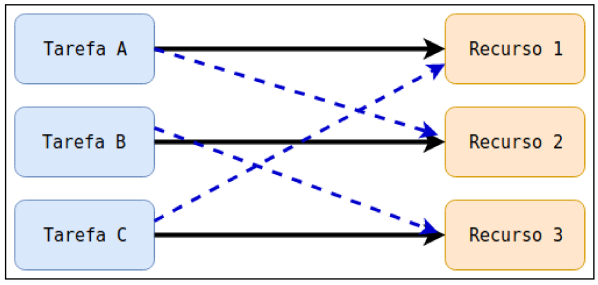
Fonte: (PATHAN, s.d.)

Figura 2.4: Inversão de prioridade ilimitada.



Fonte: (PATHAN, s.d.)

Figura 2.5: Aplicação em *deadlock*, bloqueio em cadeia impede que as tarefas adquiram os recursos necessários para a execução prosseguir a execução.



Fonte: Autoria própria

do sistema foi a missão de exploração de Marte em 1997. O robô *pathfinder* sofreu diversos *resets* após iniciar a aquisição de dados meteorológicos. A demora para executar tarefas críticas, de alta prioridade, levou-as a perder seus "tempos limite", tal falha era percebida por um *watchdog timer* que executava o processo de reinicialização (PATHAN, s.d.).

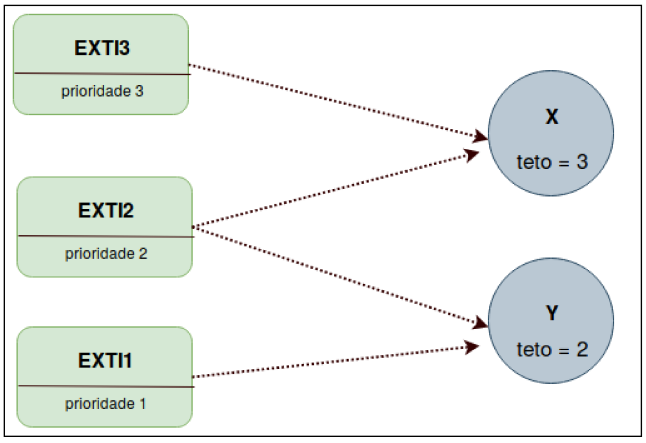
O termo *deadlock* denomina a condição particular de bloqueio de recursos aninhados, em que uma cadeia circular, de tarefas aguardando recursos, impede que todas as tarefas executam. Tarefas bloqueadas podem resultar em consequências fatais para a aplicação. Suponha, por exemplo, o seguinte conjunto de tarefas: A, B, C (Figura 2.5). A tarefa A possui o recurso 1 e aguarda pelo recurso 2; similarmente, tarefa 2 possui o recurso 2 e espera por "3", processo 3 adquiriu o recurso 3 e aguarda a liberação de "1". Nenhuma das tarefas é capaz de obter o recurso necessário para retornar a execução, portanto, a aplicação está em *deadlock*.

2.3.2 Política de acesso a recursos

O SRP é um protocolo de acesso a recursos criado para permitir que processos com diferentes prioridades compartilhem uma única pilha de execução (BAKER, 1990). Foi elaborado para impedir a ocorrência de inversão de prioridade ilimitada e *deadlocks* (BAKER, 1990). Duas definições são fundamentais para a compreensão do algoritmo: *jobs* (tarefas) e *resources* (recursos).

Tarefas são sequências finitas de instruções a serem executadas em um único processador. Normalmente são subprogramas em uma linguagem de programação qualquer, convencionalmente nomeados pela letra J seguida de índices, como: $J1$, $J2$, ..., JN. Cada tarefa pertence a uma quantidade fixa de processos, sendo um processo uma sequência finita de tarefas (BAKER, 1990).

Figura 2.6: Exemplificação do processo de cálculo do teto de um recurso.



Fonte: Autoria própria.

A critério de simplicidade, no presente texto, os termos tarefa e processo são usados de forma intercambiável. Os recursos seguem a definição apresentada acima, isto é, série de estruturas e registros não preemptivos usados de forma compartilhada.

Cada tarefa possui uma prioridade associada, $p(J)$. Prioridades são valores ordenados onde $J1$ é mais prioritário do que $J2$ caso $p(J1) > p(J2)$. Em (BAKER, 1990), para atender tanto os algoritmos estáticos quanto os dinâmicos é feita a distinção entre prioridades e níveis de prioridade ($\pi(J)$). Aqui serão sinônimos, para simplificar a discussão.

O SRP é baseado no conceito de teto de prioridades, existindo o teto de prioridade estático de cada recursos e o teto de prioridade dinâmico do sistema. Para recursos simples, única unidade, seu teto $[R]$ é calculado segundo a Equação 2.1, logo, é o nível de prioridade da tarefa mais prioritária que o reivindica (Figura 2.6). O teto dinâmico do sistema, Π, é dependente dos processos correntes e dos recursos reivindicados. Ele será o máximo entre o nível de prioridade de todos as tarefas ativas o teto de todos os recursos requeridos (Equação 2.2). A troca de contexto acontece apenas quando a tarefa J possui prioridade superior ao teto do sistema, $\Pi < \pi(J)$.

$$[R] = max(0 \bigcup \pi(J) | J \in L(R)) \tag{2.1}$$

$$\Pi = max(0 \bigcup \pi(J) \bigcup dRe | R \in Rclaimed) \tag{2.2}$$

As Equações acima (2.1 e 2.2) e, consequentemente, a plataforma RTFM são agnósticos ao modelo de priorização (estático) de tarefas adotado (RMA ou DMA). No *RTFM-kernel*, o teto de cada recurso é calculado estaticamente durante a compilação e o escalonador "rastreia" o teto dinâmico do sistema, em tempo de execução. Análises estáticas garantem que as seções críticas estão corretamente posicionadas, evitando a ocorrência de inconsistências de dados. Cabe ressaltar que, as demais propriedades (ausência de impasses e inversões de prioridade ilimitadas) são asseguradas matematicamente pelo protocolo (BAKER, 1990).

A aplicação prática dos conceitos teóricos apresentados nessa seção é melhor ilustrada no Capítulo 5. Nele, o funcionamento interno da *framework* é explorado à medida em que uma aplicação RTFM (em Rust) é construída.

2.4 Considerações finais

Ao corroborar com os ideais pregados por esse trabalho (confiabilidade e robustez), o RTFM torna-se um dos pilares de fundamentação. O formalismo estático provido pelo SRP oferece às aplicações a certeza que suas propriedades implícitas serão cumpridas, não haverá *deadlock* nem inversão de prioridade ilimitada. O programador pode, então, se ocupar com as funcionalidades de alto nível.

As particularidades envolvendo a utilização da linguagem Rust nos sistemas embarcados são melhor exploradas no próximo Capítulo (4). Nele, mostra-se como as abstrações propostas, idiomaticamente, ajudam a controlar a complexidade advinda da execução em microcontroladores. Nesse capítulo introduziu-se os motivos que levaram à adoção do RTFM, esses (e suas consequências) norteiam o restante dessa dissertação (Capítulos 5 e 6).

Capítulo 3

Rust

As linguagens de programação percorreram um longo percurso desde sua criação. Sairam de códigos de máquina para dialetos com grande poder de abstração. As últimas décadas, contudo, ficaram muito marcadas pela dicotomia entre as linguagens "antigas" e as dinâmicas, dando a ilusão de que apenas as interpretadas são mais agradáveis ao desenvolvimento. As linguagens clássicas sofreram com a passagem do tempo e não podem se modificar facilmente já que têm um longo legado a manter.

Otimizações antes da execução e a possível liberdade de sistemas de *runtime* (como um interpretador ou coletor de lixo) tornam as linguagens compiladas mais eficientes, fazendo-se necessárias em diversas situações. Aplicações microcontroladas e a computação em nuvem são bons exemplos de utilização dessa família de linguagens, justificando a recente tendência de criação de linguagens com roupagem clássica adaptadas aos novos tempos.

Os mais promissores expoentes são: Rust e Go (GOLANG.ORG, 2019); elas são linguagens ergonômicas, menos verbosas, seguras e adaptadas (PIKE, 2019), oferecendo-se como alternativa a C/C++. A forma empregada para atingir tais objetivos se difere entre os dois idiomas, interessa, ao presente trabalho, somente a linguagem Rust.

Rust é uma nova linguagem de sistema idealizada pela Mozilla. Como C e C++, fornece ao desenvolvedor um bom controle sobre o uso da memória. Entretanto, com uma estreita relação entre as operações da linguagem e o hardware, ajuda os programadores a antecipar os custos de seus códigos (BLANDY, 2015). Rust compartilha algumas ambições com C++, como as abstrações de zero custo, sumarizada por Bjarne Stroustrup: "*What you don't use, you don't pay for. And*

further: What you do use, you couldn't hand code any better.". Abstrações não precisam implicar em custos.

A linguagem Rust almeja empoderar a todos para que esses possam construir softwares confiáveis e eficientes (RUST LANGUAGE, 2019l); desempenho, confiabilidade e produtividade são as palavras chave. É *open source* e se orgulha de possuir um ecossistema democrático de desenvolvimento através de RFCs (*Request for Comments*). A infraestrutura de compilação fica a cargo da LLVM (LLVM PROJECT, 2019b).

Rust está em constante desenvolvimento e tem um fluxo de lançamento baseado em 3 canais de *release*: *Stable*, *Beta* e *Nightly*; cada qual com uma função específica. Experimentações são permitidas no canal mais instável, *Nightly*. Como o nome indica esse canal é reconstruído todas as noites, possibilitando que erros sejam descobertos o mais cedo possível. *Beta* faz o papel de transição entre a experimentação e a estabilidade. Funcionalidades chegam ao *Stable* após garantias de retrocompatibilidade com versões anteriores (RUST LANGUAGE, 2019g).

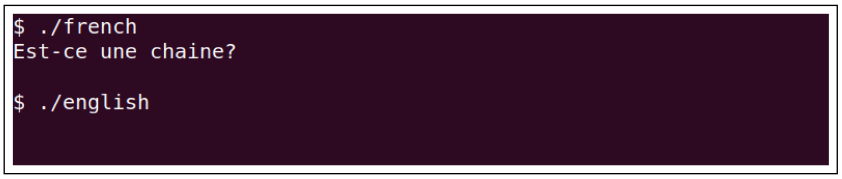
3.1 Sistemas Tipos e Segurança

Linguagem segura é aquela que protege suas próprias construções, ou seja, é capaz de garantir a integridade de suas abstrações e das operações inseridas pelos programadores usando definições idiomáticas (PIERCE, 2002). Suponha, por exemplo, uma linguagem que ofereça vetores como estrutura de dados. Um programador usando-a espera que somente as devidas posições reservadas por um vetor lhe sejam acessíveis. Acesso fora da fronteira devem emitir uma indicação de falha.

Em uma linguagem segura, as abstrações podem ser acessadas de forma realmente "abstrata". Em uma linguagem insegura as operações não são confiáveis, não se tem certeza do que irá acontecer. É necessário entender profundamente como ela se comporta (PIERCE, 2002) e a utilização equivocada pode resultar no temido comportamento indefinido (BLANDY, 2015). Pela definições acima, Rust é uma linguagem segura.

As Listagens 3.1 e 3.2 demonstram que C++ não é uma linguagem segura. Os programas se diferem apenas na frase inserida na primeira posição do vetor *strings*. No primeiro exemplo a expressão "é uma string?" está em francês e no segundo em inglês. É imaginável que o resultado para os dois casos seja equivalente: a frase acima impressa no relativo idioma. O desfecho,

Figura 3.1: Execução dos programas english.cpp e french.cpp.



Fonte: Autoria própria.

porém, surpreende. Enquanto em francês a saída é o esperado, o programa em inglês gera um comportamento indefinido (Figura 3.1).

Listagem 3.1: french.cpp. Vetor de strings com um ponteiro apontando para a primeira string do vetor.

```
#include <vector>
#include <string>
#include <iostream>

int main (int, char *[]) {
  std::vector<std::string> strings;

  strings.push_back("Est-ce␣une␣chaine?");
  const char *first = strings[0].c_str();

  for (auto s: {"Hello", "World"}) {
    strings.push_back(s);
  }

  std::cout << first << std::endl;
}
```

Listagem 3.2: english.cpp. Demonstração de como uma diferença sutil como a troca de um texto pode influência nas abstrações de uma linguagem insegura.

```
#include <vector>
#include <string>
#include <iostream>

int main (int, char *[]) {
  std::vector<std::string> strings;

  strings.push_back("Is␣it␣a␣string?");
  const char *first = strings[0].c_str();

  for (auto s: {"Hello", "World"}) {
    strings.push_back(s);
  }

  std::cout << first << std::endl;
}
```

Para compreender o porquê, é necessário entender como a estrutura de dados *string* foi implementada em C++ e suas possíveis (des)otimizações. O erro em questão decorre da otimização

de pequenas *strings* - *Small String Optimization* (SSO) (GOODRICH, 2019). Frases curtas como "*Is it a string?*" (até 16 caracteres) são armazenadas *inline,* já a *string* "*Est-ce une chaine?*" tem 18 caracteres e não recebe essa otimização. Com a inserção das palavras "`Hello`" e "`World`" o vetor '`strings`' excedeu sua capacidade inicial e precisou ser realocado. Ao mesmo tempo, '`first`' continua apontando para o endereço "antigo". A *string* em francês não estava "local" ao vetor (já estava o *heap*), por isso, o Programa 3.1 continua funcionando normalmente. Na versão em inglês, após o *loop*, '`first`' não mais aponta para o endereço válido, resultando em comportamento indefinido. Mais detalhes em (MARK ELENDT, 2018).

Um sistema de tipos é um método sintático tratável para provar a ausência de certos comportamentos classificando as instruções de acordo com os valores que elas computam (PIERCE, 2002). Da maneira como são conhecidos hoje, os *type systems* foram formalizados na década de 1990, genericamente englobam estudos em lógica, matemática e filosofia. Aqui interessa sua aplicação à ciência da computação e as vantagens por eles obtidas. Uma linguagem estaticamente tipada é aquela em que essas análises ocorrem em tempo de compilação. As dinamicamente tipadas (como Python e Scheme) usam sinalizações de *runtime* para distinguir entre os diversos tipos de estruturas (PIERCE, 2002). Ambos os tipos são capazes de garantir a segurança da linguagem.

Rust é forte e estaticamente tipada, ou seja, em tempo de compilação, o tipo de todas as variáveis deve ser conhecido, e eles não se alteram durante a execução. É interessante notar que isso não implica em perdas de ergonomia. O compilador, observando os valores e casos de utilização, consegue inferir o tipo de grande parte das variáveis, como pode ser visto nas Listagens 3.3, 3.4, 3.5, descritas adiante.

Os sistemas de tipos estáticos são eficientes na detecção de falhas, elucidando cedo alguns erros de programação. Quando passam pelo *typechecker* os programas têm a tendência de "simplesmente funcionar" (PIERCE, 2002). São capturados não apenas erros triviais como a não conversão de *string* para número antes de obter a raiz quadrada, como também problemas conceituais mais profundos, por exemplo, as unidades matemáticas em um cálculo científico (ELLIS, 2017).

Linguagens tipadas tendem a ser mais eficientes, o compilador pode usar as diferentes representações para gerar instruções de máquina apropriadas. Além disso, o sistema de tipos é uma poderosa ferramenta de documentação do sistema, oferecendo indicações sobre o comportamento das funções e interfaces. Ao contrário dos comentários e anotações, os tipos são verificados pelo

compilador, então, tem-se garantia que não estão desatualizado (PIERCE, 2002).

3.2 Sistema de propriedades, empréstimo e tempo de vida

Programas computacionais precisam gerenciar a maneira como eles usam a memória durante a execução. Em algumas linguagens isso é responsabilidade do coletor de lixo, sistema que constantemente procura posições de memória não mais usadas. Em outras, o programador deve alocá-la (*alloc*) e liberá-la (*free*) explicitamente. Rust usa uma terceira abordagem: o gerenciamento de memória é automatizado por meio de um sistema de propriedade (*ownership*) (WRIGSTAD; CLARKE, 2011; BOYAPATI; LEE, R.; RINARD, 2002) baseado em tipagem subestrutural (*substructural typing*) (WALKER, 2005), sem custos em tempo de execução.

Ao ser criada a variável é ligada a um dono. No final do escopo de seu proprietário, ela é descatada da memória (`drop`). Esse sistema dá a Rust habilidade de ser *memory-safe*, o binário resultante não terá uso após a liberação, nem vazamentos de memória. Em C++, um modelo de *ownership* pode ser aplicado através de um padrão de design, conhecido como *Resource Acquisition Is Initialization* (RAII), não sendo idiomático à linguagem. A sobreposição entre violações de memória e problemas relacionados à segurança, tornam os códigos Rust menos susceptíveis à CVEs (*Common Vulnerabilities and Exposures*) críticas.

Por padrão, as variáveis são imutáveis. Funções puras resultam em transparência referencial (SONDERGAARD; SESTOFT, 1990). Todavia, ao contrário do funcionalismo clássico, Rust não condena a mutabilidade. Para evidenciar o comportamento volátil, a palavra chave `mut` deve anteceder variáveis, referência e parâmetros que podem ter seu valor modificado durante a execução. Na Listagem 3.3, a função `add1_print` diz receber o valor imutável `x`, enquanto internamente tenta modificar seu conteúdo, resultando no erro de compilação. Além disso, conhecendo a natureza mutável ou não das variáveis, o compilador pode gerar otimizações evitando leituras repetitivas ou voláteis, ganhando-se em performance.

Listagem 3.3: Erro de compilação decorrente da alteração de uma variável imutável.

```
1 fn add1_print(x: u32) {
2     x = x + 1;
3     println!("{}", x);
4 }
5
6 fn main() {
```

```
7       let x = 42;
8       add1_print(x);
9   }
10
11  // $ cargo run
12  //     Compiling mutable v0.1.0
13  //     error[E0384]: cannot assign to immutable argument ‘x‘
14  //     --> src/main.rs:2:5
15  //      |
16  //    1 | fn add1_print(x: u32) {
17  //      |               - help: make this binding mutable: ‘mut x‘
18  //    2 |     x = x + 1;
19  //      |     ^^^^^^^^^ cannot assign to immutable argument
20  //          error: aborting due to previous error
```

Variáveis podem ser transferidas, copiadas ou emprestadas. Tipo primários, cujo tamanho pode ser determinado em tempo de compilação, são copiados em caso de reatribuição, *copy types* ou *deep copy*. Como pode ser observado na linha 5 da Listagem 3.4. Nos demais, *move types*, durante a reatribuição a propriedade é transferida invalidando a instância anterior; *shallow copy*. A Listagem 3.5 mostra esse comportamento na prática, após a reatribuição (linha 5), o uso da varíavel 'x' resulta em erro de compilação. Por mais que seja sutil, essa particularidade tem forte influência no restante da linguagem.

Listagem 3.4: Atribuição de 'x' a 'y' não invalida a instância anterior.

```
1   pub fn ownership() {
2       // Copy type
3
4       let x = 2;
5       let y = x;
6       println!("x␣=␣{}␣e␣y␣=␣{}", x, y);
7   }
8
9   // $ cargo run
10  //     Compiling ownership v0.1.0
11  //     Finished dev [unoptimized + debuginfo] target(s) in 0.24s
12  //     Running ‘target/debug/ownership‘
13  //     x = 2 e y = 2
```

Listagem 3.5: Atribuir 'x' a 'y' transfere o ownership liberando a instância anterior.

```
1   pub fn ownership() {
2       // Move type
3
4       let x = String::from("Oi␣mundo.");
5       let y = x;
6       println!("x␣=␣{}␣e␣y␣=␣{}", x, y);
7   }
8
9   // $ cargo run
10  //     Compiling ownership v0.1.0
11  //     error[E0382]: borrow of moved value: ‘x‘
12  //     --> src/mover.rs:9:33
13  //      |
14  // 7    |     let x = String::from("Oi mundo.");
15  //      |         - move occurs because ‘x‘ has type
```

```
16 //   |             `std::string::String`, which does not
17 //                       implement the `Copy` trait
18 // 8 |     let y = x;
19 //   |             - value moved here
20 // 9 |     println!("x = {} e y = {}", x, y);
21 //   |                                  ^ value borrowed here after move
22 //     error: aborting due to previous error
```

3.3 *Borrowing System*

A operação de empréstimo é guardada por duas regras verificadas pelo *borrowing system*. São elas: a qualquer momento, você pode ter uma única referência mutável ou qualquer número de referências imutáveis; sendo que essas devem sempre ser válidas. A regra de singularidade evita que ocorra *aliasing*, a mutabilidade de um dado implica na existência de um único ponto de acesso. Já a leitura, percepção, não requer coordenação e pode ser simultânea.

Essas válidações são, normalmente, efetuadas em tempo de compilação e não resultam em sobrecargas durante a execução. Apenas para tipos baseados em mutabilidade interior (padrão de design que permite *aliasing* controlado) - *Cell* (VARIOUS CONTRIBUTORS, 2019), essa análise é postergada ao *runtime*. Graças ao tratamento homogêneo de recursos, essas regras são, também, a base do sistemas de concorrência.

3.4 Ponteiros Nulos e Referências Penduradas

Além disso, o compilador também garante que o programa resultante está livre de deferências de ponteiros nulos, referências penduradas e *buffer overflow* (BLANDY, 2015), certificando que a memória será acessada de forma segura.

Ponteiros nulos são úteis à programação, porém, trazem diversos problemas quando não são tratados corretamente. Tony Hoare, criador da referência nula, sintetizou o porquê da abordagem convencionalmente adotada ser tão suscetível a resultar em erros (TONY HOARE, 2009). Criados para permitir que algumas checagem fossem desabilitadas durante a execução (buscando maior desempenho), eles diminuem a confiabilidade da aplicação. Como todos os valores podem ser "nulos", perde-se o significado de "tipo" e não há nada que lembre o programador que ele deve testar se tal referência é válida. Lembrando que deferenciar um ponteiro nulo resulta em comportamento

indefinido.

Na interpretação Rust ponteiros e valores opcionais são dois conceitos separados, os valores opcionais são tratados através da enumeração `Option`. Esse valor não é um ponteiro e portanto não pode ser deferido. Por ser um `enum`, exige que seus padrões sejam exaustivamente testados, garantindo que o programador tem ciência da natureza facultativa dessa variável (Seção 3.7). Após efetuar as validações, o binário compilado é similar ao tradicional *Null* de C/C++ (BLANDY, 2015). Outra fonte comum de defeitos, o acesso a posições indevidas de um vetor, em Rust, sempre resulta em '*panic*', finalizando irreversivelmente o *thread* corrente, (BLANDY, 2015).

Toda referência em Rust tem uma vida útil - *lifetime* - que é o escopo no qual ela é válida. Na maioria dos casos, o *lifetime* é implícito e inferido. Na definição das funções, quando o compilador não é capaz de inferir automaticamente, o programador deve anotar genericamente o relacionamento entre os parâmetros para garantir que as referências reais serão realmente válidas. O principal objetivo do sistema de *lifetime* é assegurar que o código está livre de ponteiros pendurados, informando, em tempo de compilação, que tal referência não vive tempo suficiente, como exemplificado na Listagem 3.6. Ao final de seu escopo (linha 6), a referência 'r' não pode mais ser usada. O interno do sistema de *'lifetime'* baseia-se no conceito de *subtyping* (melhor explicado na Seção 3.8.2) definindo o tempo de vida mínimo (*supertype*) e aceitando valores superiores (*subtype*).

Listagem 3.6: O sistema de 'lifetime' evita que existam referências penduradas.Ao final de seu escopo 'r' não pode mais ser usada.

```
1  {
2      let r;
3      {
4          let x = 42;
5          r = &x;
6      }
7      println!("r: {}", r);
8  }
9  // error[E0597]: `x` does not live long enough
10 //    |
11 // 5 |          r = &x;
12 //    |          ^^^^^^ borrowed value does not live long enough
13 // 6 |      }
14 //    |      - `x` dropped here while still borrowed
15 // 7 |      println!("r: {}", r);
16 //    |                        - borrow later used here
```

3.5 Sistema de modularização

Módulos são a menor unidade de privacidade em Rust. Com um módulo é possível criar funcionalidades privadas a um escopo. Eles servem ao mesmo propósito que arquivos em C e classes em C++ ou Java. Componentes internos a um módulo podem ser público (`pub`) ou privados; por padrão são, inicialmente, privados. Um dos diferenciais do sistema de modularização Rust é permitir a criação de um nível de privacidade por unidade de compilação. A expressão `pub(crate)` torna essa propriedade pública a todos elementos dentro deste pacote e privada externamente. *Crate* é o nome usual que Rust dá aos seus pacotes, unidades de compilação.

3.6 Traços e comportamentos

Outro pilar da linguagem é o sistema de *traits*. *Trait* (traço) é o recurso que indica ao compilador as funcionalidades que um tipo deve fornecer. Eles são usados para definir comportamentos compartilhados de maneira abstrata. *Traits* são similares a interfaces em outras linguagens.

O comportamento de um tipo consiste em seus métodos e funções. Diferentes tipos podem compartilhar o mesmo comportamento, caso seja possível chamar os mesmos métodos em ambos. Isso cria uma importante modularização de alto nível. Semelhante ao sistema de classes Haskell, *traits* são usados para especificar que um genérico pode ser qualquer tipo que tenha determinado comportamento (Subseção 3.8.3).

Os traços marcadores (RUST LANGUAGE, 2019h) são usados para representar propriedades básicas de um tipo e não especificam nenhum método ou função interna. Servem a propósitos diversos como, por exemplo, para informar ao compilador que tal tipo é "copiável" (*Copy Trait*). Fundamentais ao sistema de concorrência, os *traits* `Sync` e `Send`, indicam respectivamente que um tipo pode ser seguramente transferido ou compartilhado em *threads*, sendo outros exemplos de *trait marker*s.

3.7 *Enums, closure* e iteradores

A expressividade de uma linguagem de programação é a amplitude de ideias que podem ser representadas e comunicadas através dela. Quanto mais expressiva, maior a variedade de abstrações

que ela pode ser usada para representar. As funcionalidades apresentadas nessa subseção são responsáveis por boa parte do poder expressivo encontrado em Rust, que viabilizam a manifestação de ideias de alto nível clara e eficientemente.

Os `enums` permitem a definição de um tipo enumerando seus valores possíveis. São comuns a diversas linguagens, entretanto, suas capacidades se diferem entre eles. Em Rust, eles se assemelham os *Algebraic Data Types* (ADT), comuns em linguagens funcionais. Se formam a partir da combinação de outros tipos, *product types* e *sum types*. Um "produto" é uma maneira de combinar vários valores de diferentes tipos em um, como `struct` e `tuples`. Já em "tipos de soma", em vez de combinar dois valores em um, o valor que é um tipo ou outro, como *unions*. A Listagem 3.7 mostra a criação de um `enum` simples (linha 2), usando apenas a propriedade de soma e um (o `StateLed`) mais poderoso, combinando variações e registros internos (soma e produto).

Como propriedade diferencial, os `enums` podem ser quebrados em padrões, via, por exemplo, `match` e `let`. Em uma busca exaustiva todas as variações devem ser considerados (linha 20, Listagem 3.7), o não cumprimento resulta em erro de compilação (linha 15, Listagem 3.7). Dois `enums` são particularmente úteis à linguagem, o `Option` e o `Result`.

Listagem 3.7: Criação de enumerações em Rust. 'Led' é um "tipo soma". 'State Led' combina variações com valores internos.

```
1  #[derive(PartialEq)]
2  struct Led {
3      Azul,
4      Verde,
5  }
6
7  struct StateLed {
8      On(Led),
9      Off(Led),
10     Reset,
11 }
12
13 let led_azul = Led::Azul;
14
15 // error[E0004]: non-exhaustive patterns: `Verde` not covered
16 match led_azul {
17     Led::Azul => (),
18 };
19
20 match state_led {
21     On(led) if led != Led::Azul => (),
22     Off(_) => (),
23     _ => (),
24 };
```

O `Option<T>` codifica a existência (`Some(T)`) ou não de um valor (`None`). Já o `Result<T, E>`,

é usado para propagar e recuperar condições de falha. Os tipos `T` e `E` são genéricos (Subseção 3.8.3). O primeiro representa o tipo retornado em caso de sucesso, junto à variante `Ok`. O segundo simboliza o tipo do "erro" a ser retornado em caso de falha (`Err(E)`).

*Closur*es são procedimentos anônimos. Em Rust, são funções de primeira classe, isto é, podem ser salvas em uma variável (linha 4, Listagem 3.8) e usadas como argumento de uma função. Não estão ligadas a um contexto e ao contrário das funções convencionais, têm a capacidade de capturar valores do escopo no qual estão. Em *closures* não é preciso anotar o tipo dos parâmetros de entrada e saída, elas são normalmente curtas e relevantes apenas dentro de um cenário restrito. Assim, o compilador consegue inferir os tipos dos argumentos, semelhante à inferência em variáveis (linha 3, Listagem 3.8).

Os iteradores simplificam a execução uma tarefa em um sequência. O método `iter` é responsável pela lógica de iteração sobre cada item e por determinar quando a sequência foi concluída. Em Rust, eles são *lazy*. Em outros termos, não têm efeito até que a função que os consome seja chamada. Iteradores adaptáveis, como `map` e `filter`, podem ser usados transformar o comportamento de uma sequência. Quando encadeados, mesmo formando uma cadeia complexa de operações ainda são compreensíveis. Por exemplo, na linha 8 da Listagem 3.8, itera-se sobre `v` aplicando a função ‘`(4*x)/6)`‘ a cada um de seus elementos, após isso, filtra-se apenas o elementos pares (função `is_even`[1]).

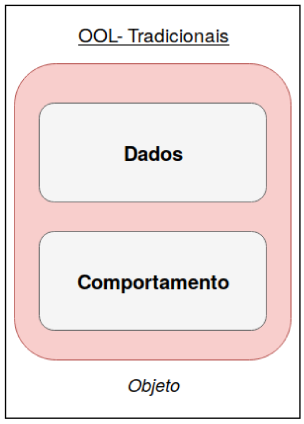
Listagem 3.8: Exemplo de utilização de 'closures' e iteradores em Rust.

```
1  fn main() {
2      let f = |x| x * 2;
3      let x2 = f(2);
4      let v = [1, 2, 3];
5      let v2 = v.iter().map(|x| ((4*x)/6) ).filter(|x| is_even(x));
6  }
7  fn is_even(x: &i32) -> bool {
8      if x % 2 == 0 {
9          true
10     } else {
11         false
12     }
13 }
```

As implementações de *Closures* e `iters` são tais que o desempenho em tempo de execução não é afetado. (RUST LANGUAGE, 2019d) compara a performance de `loops` e iteradores, reforçando o princípio de abstrações a custo zero.

[1] As expressão '`true`' (sem ";") é equivalente à declaração '`return true;`'.

Figura 3.2: Representação de um objeto em uma linguagem orientada a objetos tradicional.



Fonte: Autoria própria.

3.8 *Traits* e Orientação a objeto

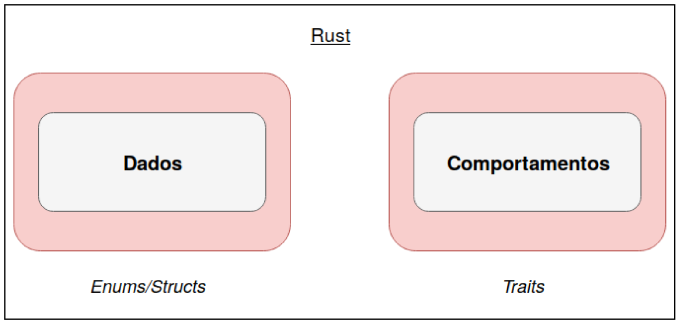
Programação orientada à objetos é um modelo de computação no qual o sistemas é construído a partir de objetos compartilhado mensagens. Originalmente concebido na década de 1960 esse paradigma surgiu como proposta para enfrentar a complexidade no desenvolvimento de software (BROOKS, 1986). Tradicionalmente, um objeto agrupa dados e procedimentos, também conhecidos como métodos (GAMMA et al., 1997), vide Figura 3.2. Não há consenso sobre quais recursos uma linguagem deve fornecer para ela ser considerada uma linguagem orientada a objetos (OOL). Entretanto, indiscutivelmente, as OOL compartilham certas características, como: conceito de um objeto, encapsulamento e herança (RUST LANGUAGE, 2019i).

Para algumas características Rust pode ser classificada como uma OOL, porém para outras não. A linguagem não aplica a definição tradicional de objetos. As *structs* e os *enums* encapsulam os dados, provendo metade das funcionalidades de um objeto tradicional (Figura 3.3) - *lightweight class*. Os métodos, ou melhor, comportamentos, são implementados através de *traits*. Essa característica influência como as funcionalidades, normalmente encontradas em OOL, são implementadas em Rust.

3.8.1 Encapsulamento

Encapsulamento em OOLs é a habilidade de esconder detalhes de implementação de forma que esses não sejam acessíveis ao códigos que usam o objeto. A única forma de interagir com essas

Figura 3.3: Objetos em Rust se diferenciam dos tradicionais pois não compartilham dados e comportamento.



Fonte: Autoria própria.

partes é através de um API pública.

Em Rust é possível controlar o encapsulamento de um componente. Como nos módulos, a palavra chave pub define quais classes, atributos, comportamentos estarão disponíveis havendo outras partes. A combinação de campos e métodos, públicos e privados, oferece o isolamento desejado para Rust ser considerada uma linguagem orientada a objeto.

3.8.2 Objetos e herança

Herança (*subclass*) é o mecanismo através do qual é possível fazer com que um objeto herde definições de um outro, obtendo dados e comportamentos. Em Rust não há herança, não é possível definir uma estrutura herdando campos e as implementação da estrutura pai (RUST LANGUAGE, 2019i). Todavia, a linguagem provêm outras maneiras de atingir os objetivos buscados com uma "herança".

A primeira razão é possibilitar o reuso de uma implementação para tipos diferentes. Em Rust, isso é atingível compartilhado a definições padrão de um método, a qual, a menos que sobrescrita, será comum por todos os objetos que implementam esse traço. A segunda, e mais importante razão, é permitir o polimorfismo, ou seja, a habilidade de substituir objetos entre si, caso que eles compartilharem certas características.

Ao invés de incentivar o reuso de códigos internos (*subclass*), o sistema de polimorfismo Rust, enaltece o compartilhamento de interfaces. Usa abstrações genéricas para possibilitar que tipo diferentes sejam intercambiáveis, contanto, que eles implementem os *traits* necessários. Isso é con-

vencionalmente conhecido como *bounded parametric polymorphism* (RUST LANGUAGE, 2019i). Na Subseção 3.8.3, apresenta-se a relevância do tópico a esse trabalho.

O artigo (MIKHAJLOV; SEKERINSKI, 1998), formaliza os problemas enfrentados com o uso da herança tradicional, popularmente conhecido como "*Fragile Base Class Problem*", respaldando a escolha da linguagem Rust pelo *subtyping*.

No *subtyping,* a generalização é em termos de interface, não de classes. Uma vez definida a relação entre os tipos, como: A é um subtipo de B, qualquer função ou procedimento aplicável a um elemento de B (*supertype*) pode ser aplicado em A (*subtype*). As implicações e detalhamentos do sistema de *subtyping* e *variances* estão além do escopo deste trabalho, cabe dizer que esses influenciam profundamente as análises do compilador. Mais informações podem ser encontradas no livro de Rust avançado, o *Nomicon* (RUST LANGUAGE, 2019m).

3.8.3 Monomorfismo

Das discussões sobre orientação a objeto, interessa ao presente trabalho sua influência na criação de interfaces genéricas. O polimorfismo estático (ou falso) é realizado durante a compilação, sendo mais conhecido como genéricos. Essa ferramenta permite que tipos abstratos sejam usando no lugar de tipos concretos durante a codificação. Com eles é possível expressar um comportamento abrangente, sem se preocupar como o tipo concreto que estará no lugar ao compilar e executar o código (RUST LANGUAGE, 2019m). Semelhante à maneira como funções usam valores desconhecidos para executar o mesmo código, em Rust, elas podem, em tempo de compilação, receber parâmetros genéricos em vez de tipos concretos (RUST LANGUAGE, 2019m).

Os genéricos em Rust se assemelham aos *templates* em C++. São ferramentas para permitir o reuso de código, evitando o tedioso e perigoso processo de duplicação manual. Eles possibilitam a codificação independente de um tipo específico e podem ser vistos como um *blueprint* ou fórmula para criar "classes" ou funções genéricas.

É similar a declarar um parâmetros na assinatura de uma função para que esse possa ser usado em seu corpo. Para usar um tipo abstrato, é necessário declará-lo no protótipo da função. Sintatimente, significa criar um identificador dentro de colchetes angulares, <>, entre o nome da função e a lista de parâmetros, conforme Listagem 3.9. Esse código pode ser traduzido como "a função `imprimir` é genérica sobre algum tipo `T`".

Listagem 3.9: Protótipo da função 'imprimir'. Ela é genérica sobre o tipo 'T'.

```
// Por convencao os tipos genericos possuem nomes curtos,
// normalmente, uma unica letra. Sao escritos em CamelCase.
// Sendo 'T', a escolha padrao da maioria dos programadores.
fn imprimir<T>(obj: T) {
    println!("{}", obj);
}
```

Note, entretanto, que ao tentar usa a definições da Listagem 3.9, no Código 3.10, tem se um problema. O compilador rejeita o código, informando que internamente espera-se que `obj` seja um objeto "imprimível". Como foi falado, para substituir objetos entre si é necessário que eles compartilhem certas características. E no caso da Listagem 3.10, o compilador não possui informações suficientes para garantir que `T`, realmente, cumpra o que dele se espera, isto é, "ser imprimível". Neste momento entra o polimorfismo parametrizado por fronteiras.

Listagem 3.10: Esse código demonstra o erro de compilação resultante ao tentar usar a definição da função 'imprimir' sem fronteiras.

```
fn imprimir<T>(obj: T) {
    println!("{}", obj);
}

fn main (){
    let num: u32 = 42;

    imprimir(num);
}

// $ cargo run --example generic-use
//     Compiling generics v0.1.0
//     error[E0277]: `T` doesn't implement `std::fmt::Display`
//     --> examples/generic-use.rs:2:20
//      |
//   2 |  println!("{}", obj);
//     |                 ^^^ `T` cannot be formatted with the default formatter
//     |
//  = help: the trait `std::fmt::Display` is not implemented for `T`
//  = note: in format strings you may be able to use `{:?}`
//                                    (or {:#?} for pretty-print) instead
//  = help: consider adding a `where T: std::fmt::Display` bound
//  = note: required by `std::fmt::Display::fmt`

// error: aborting due to previous error
```

Para os genéricos funcionarem corretamente, o comportamento esperado deve ser explícito na assinatura. Adaptando o Código 3.10 acima tem-se a Listagem 3.11. Note que está nítido que, para ser aceito por `imprimir`, `T` deve implementar o comportamento `Display`. Ao usar `imprimir` em um objeto que não compartilha o traço desejado, tem-se novamente um erro de compilação (`enum State`, Listagem 3.12).

Essa subseção tratou da utilização do polimorfismo estático em funções. Todavia, eles não

estão limitados a isso. Aplicações de genéricos em estruturas, *enums* e *traits* surgirão ao longo do texto.

Listagem 3.11: Adicionando o traço 'imprimível' como indispensável a qualquer tipo genérico 'T'.

```
use std::fmt::Display;

fn imprimir<T: Display>(obj: T) {
    println!("obj: {}", obj);
}

fn main (){
    let num: u32 = 42;
    let val = false;

    imprimir(num);
    imprimir(val);
}

// $ cargo run --example generic-traits
//     Compiling generics v0.1.0
//     Finished dev [unoptimized + debuginfo] target(s) in 2.23s
//     Running `target/debug/examples/generic-traits`
//     obj: 42
//     obj: false
```

Listagem 3.12: Como enum 'State' não implementa o trait Display. Não pode ser usado na função 'imprimir'.

```
use std::fmt::Display;

enum State {
    On,
    Off,
}

fn imprimir<T: Display>(obj: T) {
    println!("{}", obj);
}

fn main (){
    let state: State = State::On;

    imprimir(state);
}

// $ cargo run --example generic-enum
//     Compiling generics v0.1.0
//     error[E0277]: `State` doesn't implement `std::fmt::Display`
//     --> examples/generic-enum.rs:15:5
//     |
//  15 |     imprimir(state);
//     |     ^^^^^^^^ `State` cannot be formatted with the default formatter
//     |
// = help: the trait `std::fmt::Display` is not implemented for `State`
// = note: in format strings you may be able to use `{:?}`
//                                    (or {:#?} for pretty-print) instead
```

O processo de generalização estático não traz penalidades de tempo durante a execução. Nessa transformação, mais conhecida como monomorfização, o compilador analisa todos os locais onde

há códigos genéricos e os substitui com os tipos concretos com os quais o código foi chamado. Para a Listagem 3.11, por exemplo, o compilador produzirá um versão da função `imprimir` para lidar com inteiro não sinalizados de 32 bits e uma para valores booleano. Isso garante que a versão da função utilizada em execução será a mais otimizada possível para o tipo usado, com a desvantagem de produzir um binário maior.

3.8.4 *Trait Objects*

A monomorfização representa a maioria dos casos de generalização em Rust, porém, nem sempre ela é suficiente. Quando o sistema descrito acima é demasiado conservador e o tipo concreto é conhecido apenas em tempo de execução, deve-se partir para o *dispatch* dinâmico.

O sistema de *traits* continua sendo a funcionalidade base. Em uma vertente um pouco diferente, através de *trait objects*. Eles são mais próximos de um objeto na definição original da orientação a objeto. Encapsulam dados e comportamento, Listagem 3.13, entretanto não são manipuláveis diretamente. Não é possível, por exemplo, adicionar dados a um *trait object* (RUST LANGUAGE, 2019m).

Listagem 3.13: Representação em 'runtime' de um 'trait object'. Disponível no módulo 'std::raw'.

```
// Um ‘trait object‘ consiste de ponteiro de 'dados'
// e um ponteiro para uma 'vtable'.
pub struct TraitObject {
    // 'Data pointer' - aponta para o endereco do dado.
    pub data: *mut (),
    // 'vtable pointer' - aponta para a implentacao
    // do trait para o tipo correspondente.
    pub vtable: *mut (),
}
```

O despacho dinâmico, também conhecido como ligação tardia (*late-binding*), ocorre em tempo de execução. Em Rust, como em várias outras linguagens, ele é implementado via `vtable`. Essa é criada e gerenciada pelo compilador que, essencialmente, mapeia o objeto para um conjunto de ponteiros. Rust usa esses ponteiros durante a execução para descobrir qual método específico chamar.

Listagem 3.14: A 'funcao_objeto' aceita qualquer objeto que implemente 'Foo'. Note o ponteiro de indireção no argumento da função.

```
trait Foo {
    fn imprimir(&self);
}
```

```
4
5  impl Foo for u32 {
6      fn imprimir(&self) {
7          println!("Eu, {}, sou um u32", self);
8      }
9  }
10
11 impl Foo for bool {
12     fn imprimir(&self) {
13         println!("Eu, {}, sou um boleano", self);
14     }
15 }
16
17 fn funcao_objeto(obj: &Foo) {
18     obj.imprimir();
19 }
20
21 fn main() {
22     let num: u32 = 42;
23     let val = false;
24
25     funcao_objeto(&num);
26     funcao_objeto(&val);
27 }
28 // $ cargo run
29 //     Compiling generics v0.1.0
30 //     Finished dev [unoptimized + debuginfo] target(s) in 0.44s
31 //     Running `target/debug/examples/trait-object-use`
32 //     Eu, 42, sou um u32
33 //     Eu, false, sou um boleano
```

O compilador precisa conhecer, em tempo de compilação, o tamanho de todos os tipos usados. Visto que, por exemplo, para passá-los como um argumento de uma função, é necessário alocar/-desalocar espaço na pilha. Por padrão, Rust, não esconde os dados atrás de ponteiros, como outras linguagens. Entretanto um *trait object* pode ser qualquer valor que implemente o traço, como uma `String` de 24 bytes, ou um `u8` de 1 byte; tornando o uso de uma indireção necessário. Colocá-los atrás de um ponteiro, `&` ou `Box<>` (ponteiro para o *heap*, (RUST LANGUAGE, 2019b)), significa que o tamanho do tipo não é relevante, apenas o tamanho do próprio ponteiro.

Há penalidades de tempo em execução, o despacho tardio também previne que o compilador otimize o código para funções *inline* (RUST LANGUAGE, 2019m). Um 'objeto de traço' pode ser obtido a partir de ponteiro para um tipo concreto que implementa o *trait*, lançando-o como `&x as &Foo`; ou coagindo-o, `&x` usado como argumento para uma função que espera `&Foo`, vide código 3.14.

Listagem 3.15: Operações que o compilador faz para aceitar um 'trait object'.

```
1  let num: u32 = 42;
2  let val: bool = false;
3
4  // let num_t: &Foo = &num;
5  let num_t = TraitObject {
```

```
    data: &num,
    vtable: &Foo_for_u32_vtable
};

// let val_t: &Foo = val;
let val_t = TraitObject {
    data: &val,
    vtable: &Foo_for_bool_vtable
};

// num_t.imprimir();
(num_t.vtable.imprimir)(b.data);

// val_t.imprimir();
(val_t.vtable.imprimir)(y.data);
```

Internamente, o compilador executa passos semelhantes ao Código 3.15, lembrando que as operações são transparentes ao programador. Para manter as regras do sistema de *ownership*, quando necessário, o compilador chamará o destruidor correspondente (informação armazenada na `vtable`). Além disso, ele também se responsabiliza pelo tamanho e alinhamento dos dados.

Nem todos os traços podem ser usados para construir um *trait object*, é necessário que ele seja '*object-safe*'. Um *trait* é seguro se ele não requer conhecimento prévio do tamanho do objeto, `Self: Sized`; e se todos os seus métodos são seguros. Um método seguro não pode depender de propriedades do tipo concreto ao qual é implementado, `Self`.

3.9 *Macros*

Quando as composições acima não são suficientes, as abstrações sintáticas se fazem necessárias, com elas é possível ampliar os recursos da linguagem. São tão importantes quanto as abstrações de dados e funções (NORVIG, 1996). Em alguns casos, são fundamentais para garantir códigos verdadeiramente concisos e resumidos.

M*acros,* em linguagens de programação, denotam processos que ocorrem em tempo de compilação. Graças à sua natureza prematura, elas podem capturar padrões de reutilização de código que abstrações em tempo de execução não podem. Em Rust, as *macros* vêm em duas vertentes: declarativas e procedurais. Por convenção, sempre terminam com "!", diferenciando-se visualmente das funções.

A *macro* expansão em Rust se dá após a construção da *Abstract Syntax Tree* (AST). As novas construções passam pelos mesmos validações que os códigos manuais, portanto, devem atender às regras sintáticas e semânticas.

3.9.1 *Macros* declarativas

As macros declarativas, também conhecidas como "*macros* por exemplo" (KOHLBECKER; WAND, 1987), são a forma mais utilizada de metaprogramação em Rust. Em concepção, se assemelha ao `match`. Recebem uma expressão, comparam a um padrão e executam o código associado. Quanto a implementação, é suficiente aceitar que o compilador irá lidar com toda a cadeia de operações, disponibilizando o procedimento `macro_rules`. As *macros* declarativas são higiênicas, garantido que as construções não irão se sobrepor.

3.9.2 *Matching*

Similar às definições de um `match`, uma *macro* declarativa é criada a partir de uma série de regras. A Listagem 3.16 exemplifica esse processo de criação, com a definição de dois "braços" de correspondência. Considere, por exemplo, o primeiro caso descrito - `(x => $e:expr)` - ele irá corresponder a qualquer árvore sintática que atenda a estrutura `(x => <expressão-Rust>)`; como `x => 2+3` ou `x => f(42)`.

Listagem 3.16: Casos e padrões de uma macro declarativa.

```
macro_rules! foo {
    // ( x => <expressao-Rust>)
    (x => $e:expr) => { ... };
    // ( <expressao-Rust>, <expressao-Rust>, ... )
    // ( 2+2, foo(42), ... )
    ( $( $x:expr ),* ) => { ... };
}

// ‘foo!( y => 2+3 )‘ ou ‘foo!( y -> 2+3 ), por exemplo,
// resultam em erro de compilaÃ§Ã£o.
```

O símbolo $ norteia a criação de "metavariáveis", separando os identificadores próprios da *macro* dos demais existentes. Para o exemplo acima, "`$e`" recebe qualquer expressão Rust colocada após o símbolo `=>`. Os demais símbolos de um *'matcher'*, precisam corresponder exatamente à nova sintaxe. Todas as metavariáveis são acompanhadas por um fragmento de especificação, interpretação em qual forma sintática elas devem corresponder. Além de expressões (*expr*), são permitidos:

- *ident*: identificador. Exemplo: `x`, `foo`.
- *path*: nome qualificado. Exemplo: `T::SpecialA`.

- *ty*: tipo. Exemplo: `i32`, `Vec<(char, String)>`, `&T`.
- *pat*: padrão. Exemplo: `Some(t)`, `(17, 'a')`, `_`.
- *stmt*: declaração. Exemplo: `let x = 3`.
- *block*: sequência limitada de declarações. Exemplo: `log(error, "hi")`, `return 12;` }.
- *item*: item. Examples: `fn foo() { }`, `struct Bar;`.
- *meta*: "meta item", como atributos de compilação. Exemplo: `cfg(target_os = "windows")`.
- *tt*: árvore de símbolos.

As operações `(...)*` e `(...)+` são estruturas repetição, anunciam que tal padrão pode se repetir na entrada. Para esses casos, a interpretação de uma metavariável se difere das circunstâncias anteriores. Na Listagem 3.16, `$x` representa uma coleção de valores, todos para os quais esse padrão foi verdadeiro.

3.9.3 Expansão

Para cada padrão de correspondência, há um o código a ser expandido. Eles podem realizar diversas operações, como inserir o valor `$e` em um vetor ou criar uma `struct` de nome '`$e`', vide *macro* `bar` na Listagem 3.17. As estruturas de repetição, aqui, efetuam o processo contrário. Fazem com que o código interno ao `$(...)*` seja repetido para cada "sintaxe" colecionada na entrada.

Listagem 3.17: Definição de uma macro declarativa.

```
macro_rules! bar {
    ( $e:expr, $i:ident ) => {
        // Criando um estrutura de nome ‘$i‘.
        pub struct $i;

        // Inserindo ‘$e‘ no vetor ‘temp_vec‘.
        temp_vec.push($e);
    };
    ( $( $x:expr ),* ) => {
        // Para cada ‘$x‘ ‘push it‘ no vetor ‘temp_vec‘.
        $(
            temp_vec.push($x);
        )*
    };
}

//Uso:
bar!(42, Name);
bar!(42, 55, 3*5);
```

3.9.4 *Macros* procedurais

As *macros* procedurais são "funções" que aceitam um código Rust como entrada, operam nele e produzem um código como saída. Como são menos abstratas, por questões didáticas, elas serão abordadas nos capítulos seguintes.

3.9.5 *Unsafe* Rust

Rust é a união de duas linguagens *safe* Rust e *unsafe* Rust *(RUST LANGUAGE, 2018a).* As análises e discussões acima se referem ao *safe* Rust sublinguagem que herda o nome principal. Em algumas situações a análise estática é demasiada conservadora rejeitando códigos corretos. As vezes é necessário interfacear com outras linguagens ou diretamente com o hardware. Ocasionalmente é preciso extrair mais performance da aplicação. São essas as principais situação de utilização de *unsafe* Rust (RUST LANGUAGE, 2018c).

Ao usar a palavra chave *unsafe* o desenvolvedor se sobrepõe ao compilador, informando que tem consciência de suas ações e, portanto, algumas checagem podem ser desabilitadas. Interagir diretamente com funcionalidades inseguras não é recomendado, sendo necessário verificar a documentação para garantir que os contratos esperados são atingidos. Deve-se abstrair as incertezas através de uma API segura. Mais detalhes sobre a formalização semântica da linguagem Rust podem ser encontradas em: (WANG et al., 2018; JUNG et al., 2017).

3.10 Considerações finais

Alguns defendem que a linguagem Rust é muito explícita e, por isso, mais esforço cognitivo é necessário para codificar nela. Todavia, essa situação pode ser vista por outra ótica. Rust obriga os programadores a exporem suas convicções e traz para si o controle da complexidade. A responsabilidade de garantir que as propriedades não se entrelacem erroneamente fica a cargo do compilador, livrando os desenvolvedores dessa incumbência. A busca de métodos para tornar o desenvolvimento de software mais confiável é, atualmente, objeto de estudo de vários domínio da computação, a este trabalho interessa suas implicações nos softwares embarcados.

Complementar a todas as vantagens técnicas enumeradas, Rust tem uma ótima documenta-

ção, um compilador amigável com mensagens de erro úteis, um gerenciador de pacotes (*Cargo*), ferramenta de construção e documentação em código nativa. Suporta diversos editores com autoformatação e inspeção de tipos, além de possuir um formatador automático de código (*fmt*).

Capítulo 4

Rust em microcontroladores

Os primeiros microcontroladores construídos, no início da década de 1970, realizavam apenas aritmética básica. A evolução aconteceu quando os projetistas da Intel criaram o primeiro microprocessador de propósito geral. Se programado corretamente, ele podia executar diversas tarefas (K M BHURCHANDI; A K RAY, 2013). A capacidade de reuso de hardware promoveu uma revolução, já que o custo de desenvolvimento de software é consideravelmente menor (WOLF, 2012).

Quando se deseja um microcomputador completo, o microprocessador não é suficiente. É necessário complementá-lo com uma série de periféricos. Sejam genéricos como memória volátil, memória não volátil, dispositivos de entrada e saída ou especiais como controlador de interrupção, *timers* programáveis, conversor analógico digital, etc. O microcontrolador (MCU) é um microcomputador construído em um único chip, (GODSE, A. P.; GODSE, D. A., 2008), em geral com esses periféricos adicionais incorporados.

Escolher qual microcontrolador usar, envolve uma série de concessões. O aumento da quantidade de memória ou do poder computacional, normalmente, implica em maior gasto energético e preço ao consumidor final. Não há espaços para sobras. Os softwares para esses computadores devem seguir a mesma tendência e não consumir mais do que o necessário.

Para o presente trabalho, o MCU escolhido foi o STM32L052K6, da STMicroelectronics. A família STM32L0 é composta por microcontroladores de 32 bits com consumo de energia ultrabaixo. A combinação de um núcleo Arm® Cortex®-M0+ e recursos de ultra baixa potência, o torna apropriado para aplicações que funcionam a bateria (STMICROELECTRONICS, 2019). Com apenas 8 Kbytes RAM e 32 Kbytes de *flash*, o *firmware* para essa plataforma deve usar os recursos de forma

eficiente.

A interface com os módulos de entrada/saída, em microcontroladores baseados em Arm® Cortex®-M, é feita através de mapeamento de memória (*memory mapped I/O*). Nessa API em hardware, os registros dos periféricos são atribuídos a posições arbitrárias de memória. Como vantagem, eles podem ser manipulados por meio de instruções normais de carregamento e armazenamento (*load/stores*). Os endereços de atuação são dependentes do fabricante e da família em questão. O mapa de um dispositivo específico é disponibilizado em seu manual de referência. A Figura 4.1, traz a estrutura de endereçamento do STM32L052, por exemplo.

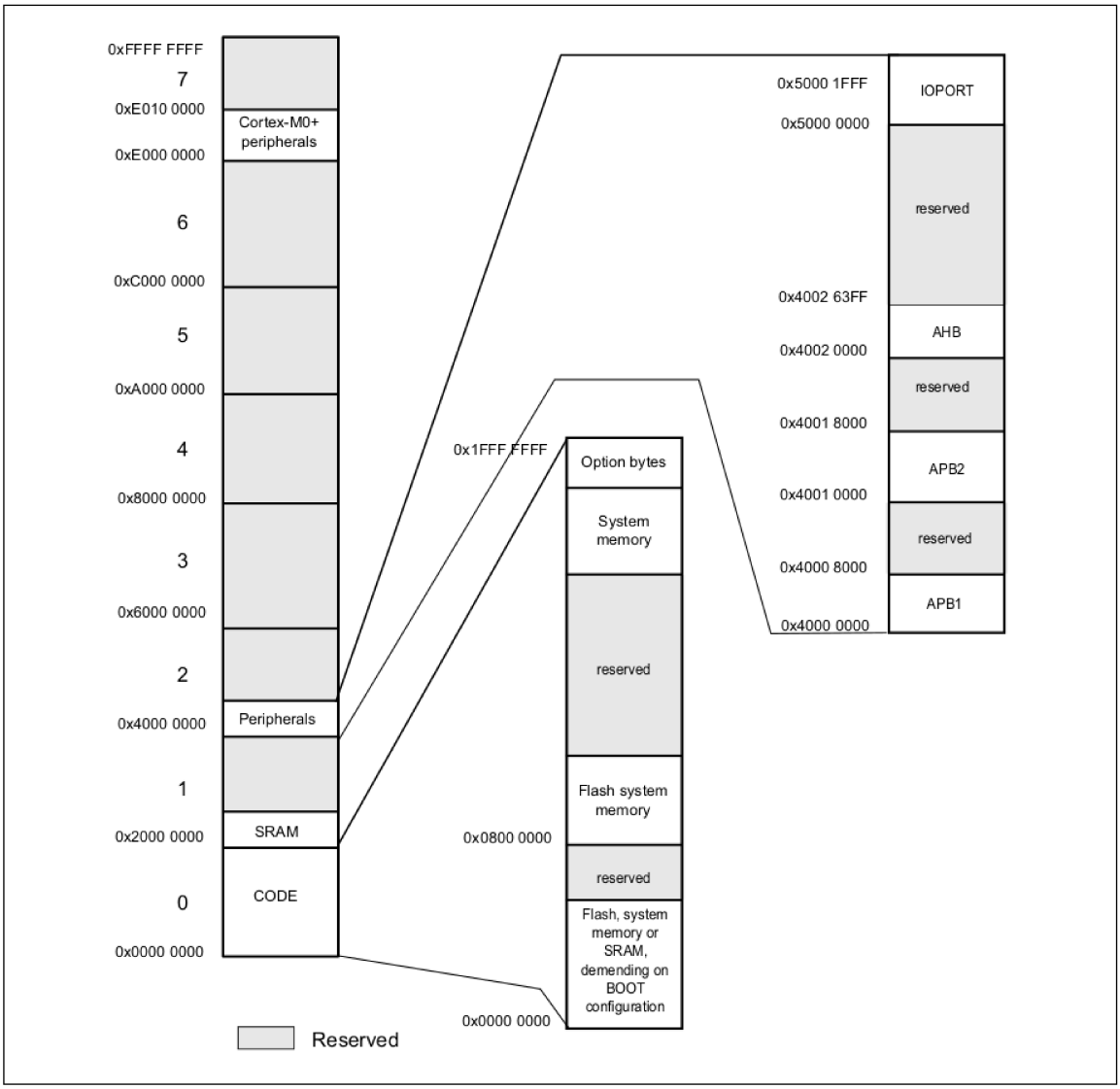
Aplicações para sistemas embarcados não são novidade na comunidade Rust. Desde sua versão 0.12, em 2014, provas de conceito, como RustyPebble (BALDASSARI, 2014), vêm sendo criadas. Entretanto, apenas em 2018, a linguagem passou a suportar oficialmente a compilação para microcontroladores (APARICIO, 2018d), resultado de um trabalho colaborativo que envolveu diversos times do ecossistema (APARICIO, 2018c).

4.1 *Core* e Biblioteca Padrão

Parte das dificuldades advinham de como as funcionalidades foram divididas em Rust. Enquanto o *core* se preocupa com concepções agnósticas à plataformas, a biblioteca padrão (*standard library*) se responsabiliza por interfacear como o sistema operacional. Fortemente ligada, por exemplo, à *libc*, ela controla a alocação dinâmica de memória, criação de *threads*, comunicação em rede. Além de controlar os periféricos de alto nível, a *standard library* é responsável pela inicialização do programa, isto é, de maneira simplória, cuida dos passos iniciais até chegar à função '`main()`'.

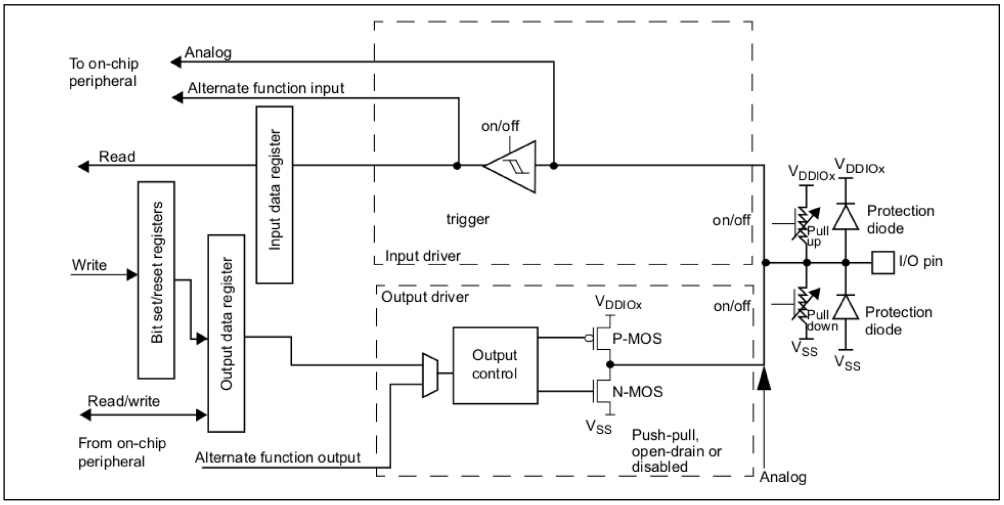
Os *firmwares* possuem funcionalidades restritas e normalmente não contam com um sistema operacional de propósito geral, inviabilizando a utilização da biblioteca padrão. Até o ano passado (2018), a interdependência entre o *core* e a *std* restringia o desenvolvimento de binários `#![no_std]` (independentes da biblioteca padrão) ao canal *Nightly*. Não era possível, por exemplo, controlar o fluxo de inicialização da aplicação ("indicar uma `main` própria") ou definir uma rotina de '`panic`' apropriada no canal *Stable*. Além disso, por muito tempo, o processo de compilação dessas soluções exigia o uso de ferramentas não oficiais, como um sistema de construção alternativo, o *crate* `Xargo` (APARICIO, 2018d).

Figura 4.1: Mapa de memória do microcontrolador STM32L052Cx, extraído do manual referencial da plataforma.



Fonte: (STMICROELECTRONICS, 2017)

Figura 4.2: Estrutura básica de um pino de entrada/saída, extraído do manual referencial da plataforma.



Fonte: (STMICROELECTRONICS, 2017)

Ao longo deste capítulo serão apresentados os principais conceitos associados à utilização da linguagem Rust em microcontroladores, resultando na biblioteca de abstração de hardware proposta para o MCU escolhido. Os detalhes relacionados às configurações de runtime e demais inconvenientes '`no_std`' serão ignorados a princípio e, devidamente, discutidos na Subseção 4.4.3.2.

4.2 Hello World em Rust

Os pinos de entrada e saída em um microcontrolador da família STM32L0 têm o seguinte esquemático, apresentado na Figura 4.2. Nota-se que eles possuem diferentes propósito e topologias. Podem se comportar como entrada/saída digital, entrada análogica ou função alternativa.

O objetivo desta seção é acender um led, portanto, o pino será configurado como saída digital. A topologia elétrica de saída é variável, podendo ser coletor aberto (*open-drain*) ou *push-pull*. Para essa aplicação será *push-pull*. A critério de simplicidade, as configurações de energia e *clock* serão ignoradas.

O primeiro passo para usar um periférico é apontar para os seus registros, uma vez que manipular um periférico significa escrever/ler em seus registros. Recorrendo à referência técnica do microcontrolador, têm-se as posições de interesse do GPIOA (Figura 4.3). A Listagem 4.1, é uma primeira tentativa de acender um led em Rust, sendo melhorada nas seções seguintes. Note

que a primeira linha (Listagem 4.1) indica que essa aplicação não depende da biblioteca padrão (`#![no_std]`), explícito ou não, esse fato é verdadeiro para os demais códigos apresentados nesse capítulo.

Listagem 4.1: Acendendo um led de forma rudimentar.

```
#![no_std]

use core::ptr;
use core::mem::size_of;

struct GpioA;

// let _: size_of::<GpioA>();    // == 0

impl GpioA {
    const MODE_REG: *mut u32 = 0x5000_0000 as _;
    const OTYPE_REG: *mut u32 = 0x5000_0040 as _;
    const OD_REG: *mut u32 = 0x5000_0014 as _;

    fn new () -> GpioA {
        GpioA
    }

    fn into_output(&mut self, pin: u8) {
        let out_mode = 0b01;
        unsafe {
            let output = (ptr::read_volatile(Self::MODE_REG)
                          & !(0b11 << pin))
                | (out_mode << pin);
            ptr::write_volatile(Self::MODE_REG, output);
        }
    }
    fn into_input(&mut self, pin: u8) {
        // Codigo omitido.
    }

    fn push_pull(&mut self, pin: u8) {
        // Codigo omitido.
    }

    fn set_high(&mut self, pin: u8) {
        // Codigo omitido.
    }

    fn is_low(&self) -> bool {
        // Codigo omitido.
    }
}

fn main() {
    let mut pin_a = GpioA::new();

    // transformando o pino 5 do GPIOA em saida.
    pin_a.into_output(5);

    // configurando a topologia da porta em push-pull.
    pin_a.push_pull(5);

    // alterando o nivel logico do pino para 1.
    pin_a.set_high(5);
}
```

Figura 4.3: Informações a respeito do GPIOA.

Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
IOPORT	0X5000 1C00 - 0X5000 1FFF	1K	GPIOH	*Section 9.4.12: GPIO register map*
	0X5000 1400 - 0X5000 1BFF	2 K	Reserved	-
	0X5000 1000 - 0X5000 13FF	1K	GPIOE	*Section 9.4.12: GPIO register map*
	0X5000 0C00 - 0X5000 0FFF	1K	GPIOD	*Section 9.4.12: GPIO register map*
	0X5000 0800 - 0X5000 0BFF	1K	GPIO C	*Section 9.4.12: GPIO register map*
	0X5000 0400 - 0X5000 07FF	1K	GPIOB	*Section 9.4.12: GPIO register map*
	0X5000 0000 - 0X5000 03FF	1K	GPIOA	*Section 9.4.12: GPIO register map*

Fonte: (STMICROELECTRONICS, 2017)

4.2.1 *Zero Sized Types*

Os *Zero Sized Types* (ZST), ou *phantom types*, são tipos especiais que não ocupam espaço de memória, existem apenas durante as análises estáticas. O compilador reduz as operações que produzem ou armazenam ZST em *no-op* (nenhuma operação) (RUST LANGUAGE, 2019e).

A estrutura "`GpioA`" é um ZST. E, como indicado pelo nome, agrupará as funcionalidades da porta de entrada e saída "A". O bloco `impl GpioA` implementa o comportamento do periférico. As constantes `MODE_REG`, `OTYPE_REG` e `ODR_REG` armazenam o endereço (ponteiro) dos registros homônimos, como pode ser observado na Listagem 4.1 (linhas 11-13). Como prova de conceito, somente a função `new` e os métodos `into_output`, `into_input`, `push_pull`, `set_high` e `is_low` foram criados. Note que o método `is_low` (linha 40) é o único a receber um referência imutável, indicando ser o único que não altera o valor dos registros. Como os ponteiros são mutáveis e globais a todos os métodos, as leituras e escritas são voláteis, inibindo otimizações.

O método `into_output` (linha 19) efetuam os passos necessário para configurar um '`pin`' como um saída digital. Conforme descrito no manual de referência do dispositivo (STMICROELECTRONICS, 2019), escreve (linha 25) '`01`' (em binário) na posição pertinente ao "pino" recebido (linhas 23 e 24) do registro de configuração de modo (`MODE_REG`). Em Rust, as funções `read_volatile` e `write_volatile` equivalem, em funcionalidade, à palavra chave '`volatile`' em C.

Por mais que seja funcional, o Programa 4.1 possui abstrações rudimentares e pouco confiá-

veis. Observe na Listagem 4.2, por exemplo, como o uso equivocado das funções `into_output` e `into_input` resulta em comportamento indefinido. Inicialmente o pino PA5 foi configurado como saída/*push-pull* (linhas 20 e 23) e depois como entrada dentro da "função complicada" (linhas 25 e 9-13). Abstratamente, não há garantias que o método `set_high` (linha 28) executado após a '`funcao_complicada`' realmente terá o efeito esperado. Pelo tamanho do código mostrado, esse caso é um erro trivial e fácil de ser percebido. Todavia, a medida em que o programa cresce em complexidade erros como esse, se tornam sutis e difíceis de capturar.

Listagem 4.2: A abstração insegura. Há a possibilidade gerar de "comportamento indefinido".

```
1  use core::ptr;
2
3  struct GpioA;
4
5  impl GpioA {
6      // Codigo omitido.
7  }
8
9  fn funcao_complicada(port_a: &mut GpioA, pin: u8){
10     // Qualquer funcionalidade complexa
11     // cuja compreensao requer esforco.
12     port_a.into_input(pin);
13 }
14
15 fn main() {
16     let pin5 = 5;
17     let mut port_a = GpioA::new();
18
19     // transformando o pino 5 do GPIOA em saida.
20     port_a.into_output(pin5);
21
22     // configurando a topologia da porta em push-pull.
23     port_a.push_pull(pin5);
24
25     funcao_complicada(&mut port_a, pin5);
26
27     // alterando o nivel logico do pino para 1.
28     port_a.set_high(pin5);
29 }
```

Alem disso, a estrutura `GpioA` pode ser instanciada mais uma vez criando *aliasing* aos registros em questão. Na Listagem 4.3, por exemplo, o pino 5 (porta A) foi erradamente "reinstanciado" (e configurado) na '`funcao_qualquer`' (linha 14), o mesmo poderia acontecer com qualquer outro periférico. Com a atual abstração, o hardware é um grande estado compartilhado e mutável, contrariando o princípio de unicidade tal pregado pela linguagem Rust. Ademais, os periféricos são diretamente controlados por ponteiros crus (*raw pointers*) e a aplicação está infestada de operações *unsafe*, como no método `into_output` (linha 21, Listagem 4.1).

Listagem 4.3: Dupla instanciação do "GpioA".

```
use core::ptr;

struct GpioA;

impl GpioA {
    // Codigo omitido.
}

fn funcao_qualquer(){
    // Criando um segunda instancia do periferico
    // GpiaA, provocando ‘aliasing‘ e comportamento
    // indefinido.
    let pin5 = 5;
    let mut port_a = GpioA::new();

    port_a.into_input();
}

fn main() {
    let pin5 = 5;
    let mut port_a = GpioA::new();

    // transformando o pino 5 do GPIOA em saida.
    port_a.into_output(pin5);

    // configurando a topologia da porta em push-pull.
    port_a.push_pull(pin5);

    // alterando o nivel logico do pino para 1.
    port_a.set_high(pin5);

    funcao_qualquer();
}
```

4.3 Abstrações seguras para acesso a periféricos

O primeiro ponto a ser corrigido é a possibilidade de criação de múltiplas instâncias. As posições de memória, apontadas pelas estruturas, controlam um periférico físico único e isso precisa ser refletido na aplicação. A técnica utilizada será a construção de um *Singleton* (GAMMA et al., 1997).

Cada hardware (funcionalidade) será protegida por uma *Option*, para garantir que será instanciado apenas um vez. No primeiro “take” (linha 26), a variável ’p’ (linha 14) recebe `Some(GpioA)`, ao abrí-la (`p.unwrap()` - linha 17) retorna-se o ’`GpioA`’ para a `main`. Ao tentar “pegá-lo” novamente (linha 30), ’`p.unwrap()`’ de um `None` resulta em `panic!`[1], como pode ser visto no Código 4.4. Esse modelo envolve um pequeno custo de execução durante inicialização do periférico, porém proporciona diversas melhorias. Além de evitar *aliasing* nos registros, sendo a instância única, é possível rastreá-la dentro do programa, permitindo validações do sistema de empréstimo

[1] A função ’`replace(<dest>, <src>)`’ retorna o atual e valor do destinatário (`dest`) e move o conteúdo de ’`src`’ para lá, assim, o erro acontece apenas a partir da segunda interação

e propriedade. Ganha-se garantias de que as regras de unicidade, mutabilidade e consumo são seguidas.

Listagem 4.4: Usando o enum "Option"para transformar os periféricos em singleton.

```
use core::ptr::replace;
struct GpioA;

impl GpioA {
    // Codigo omitido.
}

struct Peripherals {
    gpioa: Option<GpioA>
}

impl Peripherals {
    fn take_gpioa(&mut self) -> GpioA {
        let p = unsafe {
            replace(&mut self.gpioa, None)
        };
        p.unwrap()
    }
}

static mut PERIPHERALS: Peripherals = Peripherals {
    gpioa: Some(GpioA),
};

fn main() {
    let port_a = unsafe { PERIPHERALS.take_gpioa() };

    // thread 'main' panicked at 'called `Option::unwrap()`
    //   on a `None` value', src/libcore/option.rs:345:21
    let port_a2 = unsafe { PERIPHERALS.take_gpioa() };
}
```

Aproveitando-se do poder dos ZSTs, é possível incorporar outras regras de negócio ao código sem penalidades durante a execução. Embutir constantes e invariâncias da aplicação no sistema de tipos transforma erros de execução em problemas de compilação. Abstrações previamente pensadas geram códigos claros e adaptados ao problemas que se deseja solucionar.

4.3.1 *Typestate*

Enquanto o tipo de um dado determina o conjunto de operações que são permitidas em seus objetos, o *typestate* delimita o subconjunto dessas operações permitidas em um determinado contexto (STROM; YEMINI, 1986). Ele aumenta a confiabilidade do programa detectando, em tempo de compilação, sequências sintaticamente corretas mas semanticamente inválidas (STROM; YEMINI, 1986). Aqui, o *typestate* será empregado na construção das abstrações seguras de acesso aos periféricos. Ao restringir o número de operações válidas a um objeto (periférico) em deter-

minado momento, espera-se evitar os "comportamentos indefinidos" causados por configurações equivocadas.

O exemplo mais clássico de demonstração do conceito de *typestate* é a abstração de um "arquivo". Considere o objeto `MyFile`, antes de ser aberto, ele não pode ser lido e, quando fechado, não deve mais estar disponível para leitura. Embora nunca deixe de ser um "arquivo", apenas em parte da sua vida útil, a operação de leitura é permitida. Uma abstração baseada em *typestate* pode controlar esses estados e operações intermitentes, assegurando que as operações estão restritas ao intervalo desejado.

A maioria das linguagem de programação não possui suporte para tratar esse tipo de invariância. Rust, em suas versões iniciais (antes da versão 1.0) era uma linguagem com *typestate* interno (HAVERBEKE, 2017). Ao perceber que a união de *moves* e *phantom types* promovia uma maneira eficaz e idiomática de tratar a evolução do estado de um objeto, o *typestate* explícito foi retirado da linguagem.

Considere, por exemplo, o pino de propósito geral mostrado na Figura 4.2. Para evitar os problemas apresentados na subseção anterior, os estágios de composição do pino serão traduzidos para uma máquina de estados de "tipos", garantindo que a semântica de configuração será atendida em tempo de compilação. Nesse modelo, apenas as transições previamente definidas serão permitidas e para cada estágio de configuração, somente as operações pertinentes estão disponíveis.

O objetivo aqui é o mesmo da subseção anterior, acender o Led conectado ao pino PA5. Por padrão, conforme indicado no manual de referência do MCU (STMICROELECTRONICS, 2017), após o inicialização, os pinos estão em modo entrada analógico - comportamento traduzido para a Listagem 4.5 (linha 25). Para que o (atual) '`PA5<Analog>`' cumpra o esperado é necessário transformá-lo em uma saída digital (método `into_output()`) e depois configurá-lo em topologia '`push_pull()`'. Ao contrário dos códigos anteriores, na Listagem 4.5, o método '`set_high`' só estará disponível para o pino PA5 se ele estiver configurado corretamente, ou seja, se ele for um `PA5<Output<PushPull»`.

Cada estágio de configuração é representado por um "tipo fantasma" e por funções que traduzem o comportamento (real) do periférico nesse momento. A cada método de configuração executado, a instância recebida é "consumida" (*move types*) e um novo "tipo fantasma" e retornado. Por

exemplo, o método 'into_output' "consome" o PA5<Analog> e retorna um 'PA5<OutputPin>' (linha 39, Listagem 4.5). Como a topologia de um pino está ligada ao seu modo de operação, a função push_pull será um método de um OutputPin (linha 46), não de um pino qualquer. Chamá-la em um pino analógico, por exemplo, resulta em erro de compilação error[E0599] (linha 69).

Listagem 4.5: Usando o sistema de tipos para incorporar regras de negócio no código.

```
use core::ptr::replace;
// PhantomData: Sao ‘Zero-sized type‘ usados para marcar
// que uma ‘coisa‘  age como dona de ‘T‘.
// https://doc.rust-lang.org/std/marker/struct.PhantomData.html
use core::marker::PhantomData;

struct GpioA;

struct OutputPin;
struct InputPin;
// Modo default segundo o manual.
struct Analog;

struct PushPull;

struct PA5<MODE> {
    _mode: PhantomData<MODE>,
}

struct Output<TOPO>{
    _topology: PhantomData<TOPO>,
}

struct Parts {
    pa5: PA5<Analog>
}

impl GpioA {
    // registros omitidos.

    fn split (self) -> Parts {
        Parts {
            pa5: PA5 { _mode: PhantomData }
        }
    }
}

impl PA5<Analog>{
    fn into_output(self) -> PA5<OutputPin> {
        PA5 { _mode: PhantomData }
    }
    //...
}

impl PA5<OutputPin> {
    fn push_pull(self) -> PA5<Output<PushPull>> {
        PA5 { _mode: PhantomData }
    }
}

impl<MODE> OutputPin for PA5<Output<PushPull>> {
    fn set_high(&mut self) {
        unsafe { (*GPIOA::ptr()).bsrr.write(|w| w.bits(1 << 5)) }
    }
}
```

```
56
57 // Tratamento dos perifericos conforme o codigo anterior.
58 // struct Peripherals { .. }
59 // impl Peripherals { .. }
60 // static mut PERIPHERALS ... ;
61
62 fn main() {
63     let port_a = unsafe { PERIPHERALS.take_gpioa() };
64
65     let gpioa = port_a.split();
66
67     // error[E0599]: no method named ‘push_pull‘ found for type
68     // ‘PA5<Analog>‘ in the current scope
69     let _ = gpioa.pa5.push_pull();
70
71     let led = gpioa
72         .pa5.into_output()
73         .push_pull();
74
75     // error[E0382]: use of moved value: ‘gpioa.pa5‘
76     // note: move occurs because ‘gpioa.pa5‘ has type ‘PA5<Analog>‘,
77     // which does not implement the ‘Copy‘ trait
78     let led2 = gpioa
79        .pa5.into_input();
80 }
```

Construir objetos através da transformação de outros é o padrão de design, o *Builder Pattern* (GAMMA et al., 1997). Graças à semântica de movimento, os construtores são adaptáveis, assim, não é preciso criar diversos construtores com diferentes argumentos (VARIOUS CONTRIBUTORS, 2018). Um pino, por exemplo, não pode ter duas configurações, a transformação `into_output` consome '`PA5`' (`self`) que não poderá mais ser usado. A dupla configuração, vista na seção anterior (4.4), aqui o resulta em erro de compilação, `error[E0382]` (linha 78). Criou-se, assim, um sistema de *lock* em software. Isso foi apenas um exemplo, a ideia pode ser (e será na Subseção 4.4.4.1) ampliada para tratar mais regras ou funcionalidades.

4.3.2 *Macros*

Para a abstração mostrada na Listagem 4.5 funcionar corretamente é necessário separar o '`GpioA`' pinos independentes. Até aqui, visando a didática do texto, a função '`split`' (linha 31) retorna apenas o pino 5 (linha 25) e, esse foi o único a ser configurado. A implementação dos estágios de configuração prevê que a máquina de estados ilustrada no '`PA5`' seja repetida para todos os outros pinos, uma tarefa demorada e suscetível a erros.

A *macro* declarativa '`gpio!`' (Código 4.6) foi construída para contornar esse problema. Nela, o *matcher* (linhas 18-20) irá absorver as identidades, expressões e tipos pertinentes e produzirá o *typestate* de configuração para cada "pino" recebido. Note que as linhas 21 a 50 são a tradução "em

macros declarativa" dos passos apresentados na Listagem 4.5 (linhas 24-49), a estrutura 'Parts' (linha 21), por exemplo, é populada com todos os $pxi recebidos (linha 25) e assim por diante. No techo - linhas 59 a 63 - mostra-se como "executar" essa *macro* para os pinos PA4 e PA5.

Listagem 4.6: Abstrações para os pinos 4 e 5. Usando macros incrementais.

```
1  use core::ptr::replace;
2  use core::marker::PhantomData;
3
4  struct GpioA;
5
6  struct OutputPin;
7  struct InputPin;
8  // Modo default segundo o manual.
9  struct Analog;
10
11 struct PushPull;
12
13 struct Output<TOPO>{
14     _topology: PhantomData<TOPO>,
15 }
16
17 macro_rules! gpio {
18     ($GpioX:ident [
19         $($PXi:ident: ($pxi:ident, $i:expr, $MODE:ty),)+
20     ]) => {
21         struct Parts {
22             $(
23                 $pxi: $PXi<$MODE>,
24                 $(
25                     pub $pxi: $PXi<$MODE>,
26                 )+
27             )+
28         }
29         impl $GpioX {
30             fn split(self) -> Parts {
31                 ...;
32             }
33         }
34         $(
35             struct $PXi<MODE> {
36                 _mode: PhantomData<MODE>,
37             }
38             impl $PXi<Analog>{
39                 fn into_output(self) -> $PXi<OutputPin> {
40                     $PXi { _mode: PhantomData }
41                 }
42                 ...;
43             }
44
45             impl $PXi<OutputPin> {
46                 fn push_pull(self) -> $PXi<Output<PushPull>> {
47                     $PXi { _mode: PhantomData }
48                 }
49             }
50         )+
51     }
52 }
53
54 // Tratamento dos perifericos conforme o codigo anterior.
55 // struct Peripherals { .. }
56 // impl Peripherals { .. }
57 // static mut PERIPHERALS ... ;
```

```
58
59 gpio!(GpioA
60       [
61           PA4: (pa4, 4, Analog),
62           PA5: (pa5, 5, Analog),
63       ]);
64
65 fn main() {
66     let port_a = unsafe { PERIPHERALS.take_gpioa() };
67
68     let gpioa = port_a.split();
69
70     let led = gpioa
71         .pa5.into_output()
72         .push_pull();
73
74     let botao = gpioa
75         .pa4.into_input();
76 }
```

4.4 Abstrações, *Runtime*

As Seções 4.2 e 4.3 são prova de conceito, nelas o mapeamento dos registros foi feito manualmente e as abstrações estão simplificadas e incompletas. O excesso de trabalho braçal e falta de separação em níveis de abstração não trazem o sentimento colaborativo, tão presente na comunidade Rust. O grupo de trabalho "Rust para sistemas embarcados" (EMBEDDED RUST WORKING GROUP, 2019) veio para unir esforços e mudar o paradigma em questão.

Dentre seus objetivos está o desejo de promover experiências mais ergonômicas de desenvolvimento, semelhantes à criação de aplicações *desktop*. Enaltecendo a criação e manutenção de pacotes segundo regras de versionamento semântico (PRESTON-WERNER, 2019) e o uso de servidores de integração contínua, amparado pelo *Cargo* e seus subsistemas. O repositório (RUST EMBEDDED, 2019a) combina projetos e interesses dos envolvidos.

Desde 2018, Rust suporta oficialmente a compilação cruzada via *Cargo*. Para tal, é necessário somente que a linguagem suporte a plataforma desejada, (RUST LANGUAGE, 2019l). A instalação de uma arquitetura destino é feita via *Rustup*, (RUST LANGUAGE, 2019k), usando um comando simples (`rustup add target **`).

Nas subseções seguintes, algumas ferramentas (4.4.1) e bibliotecas (4.4.2, 4.4.3 e 4.4.4) criadas para facilitar o desenvolvimento de aplicações embarcadas em Rust serão apresentadas. Esse capítulo termina com a verificação e validação (Subseção 4.4.5) das soluções mostradas.

4.4.1 SVD2Rust

A primeira ferramenta que merece atenção é o projeto `svd2rust`. Seu propósito é transformar arquivos CMSIS-SVD (*System View Description*), disponibilizados pelo fabricante, em código Rust seguro. Os arquivos SVD descrevem as funcionalidades de hardware de um MCU, em particular, listam todos os periféricos disponíveis e suas respectivas posições de memória.

`Svd2rust` é uma ferramenta de linha de comando, que retorna em uma API segura para acessar os periféricos no dispositivo (RUST EMBEDDED, 2019e). Está disponível através de um '`cargo install`'; mais informações e instruções estão disponíveis em (RUST EMBEDDED, 2019e). O código gerado para o microcontrolador STM32L0 pode ser encontrado em (SILVA, 2018).

Como na seção anterior, os periféricos gerados estão moldados por um *singleton* e a única forma de acesso é via `Peripherals::take`, como no Código 4.8. Cada `RegisterBlock`, é uma estrutura que representa a memória pertinente a um periférico (Listagem 4.7) para o GPIOA. Cada campo no `RegisterBlock` expõe uma combinação de métodos de leitura, escrita ou modificação (`modify`), dependente do tipo do registro em questão.

Caso o registro seja somente de leitura, apenas o método `read` é gerado. Se o registro for apenas de escrita, somente `write` estará disponível. Se as duas operações forem permitidas, os três métodos serão gerados.

Listagem 4.7: Bloco de registro do 'gpioa' para o microcontrolador STM32L052.

```
pub struct RegisterBlock {
    #[doc = "0x00 - GPIO port mode register"]
    pub moder: MODER,
    #[doc = "0x04 - GPIO port output type register"]
    pub otyper: OTYPER,
    #[doc = "0x08 - GPIO port output speed register"]
    pub ospeedr: OSPEEDR,
    #[doc = "0x0c - GPIO port pull-up/pull-down register"]
    pub pupdr: PUPDR,
    #[doc = "0x10 - GPIO port input data register"]
    pub idr: IDR,
    #[doc = "0x14 - GPIO port output data register"]
    pub odr: ODR,
    #[doc = "0x18 - GPIO port bit set/reset register"]
    pub bsrr: BSRR,
    #[doc = "0x1c - GPIO port configuration lock register"]
    pub lckr: LCKR,
    #[doc = "0x20 - GPIO alternate function low register"]
    pub afrl: AFRL,
    #[doc = "0x24 - GPIO alternate function high register"]
    pub afrh: AFRH,
    #[doc = "0x28 - GPIO port bit reset register"]
    pub brr: BRR,
}
```

O método `read` lê o registro através uma única instrução de leitura volátil (LDR). Ele retorna um *proxy* `R` que permite acesso ao bits "que podem ser lidos" do registro, protegendo posições reservadas e de escrita, como no Código 4.8.

De forma semelhante, a função `write` emite uma única instrução de escrita volátil (STR). A estrutura 'W' possibilita a construção de estados válidos para o registros em questão. Apenas o construtor de `reset_value` está disponível por padrão. O restante das funcionalidades devem ser construídas no estilo do *Builder Pattern*, via função de primeira classe (*closure*), vide a Listagem 4.8. Lembrando que apenas os campos modificáveis estão disponíveis.

Na escrita de múltiplos bits, o compilador não consegue assegurar que somente posições válidas serão acessadas. O programador deve sobrepor à análise, marcando essa operação como *unsafe* (Listagem 4.8).

O método `modify` é a combinação dos dois anteriores, lê unicamente, modifica o valor e emite uma escrita única. Esse método aceita uma *closure* que especifica como o registro será modificado (o argumento `w`) e também fornece acesso ao estado do registro antes da modificação (o argumento `r`). Um exemplo de utilização pode ser visto no Código 4.8.

Listagem 4.8: Exemplos de utilização das funções gerados pela ferramenta svd2rust. Corroborando o conceito de APIs seguras.

```
fn main () {
    let mut peripherals = stm32l052::Peripherals::take().unwrap();

    // unwrap de um ‘None‘ resulta em ‘panic!‘.
    let panics = stm32l052::Peripherals::take().unwrap();

    // Setando o bit UG(Update Generation) no
    // registro EG (Event Generation) da unidade
    // de timer TIM2.
    peripherals.TIM2.egr.write(|w| w.ug().set_bit());

    // Leitura do bit 4 do registro IDR (Input Data) GPIOA
    let pin_4 = peripherals.GPIOA.idr.read().id4();

    // Modificacao do registro OTYPE (Output Type).
    // Note que o valor atual do registro foi levado em consideracao
    // pela mascara de modificacao. E que, como mais de um bit foi alterado,
    // a operacao precisa ser precedida por ‘unsafe‘. O programador
    // se responsabiliza por modificar apenas o campos permitidos.
    peripherals.GPIOA
        .otyper
        .modify(|r, w|
                unsafe {
                    w.bits(r.bits() & !(output_type << 3))
                });
}
```

Quando a funcionalidade `rt` está ativada, além das APIs acima, é gerado também o arquivo

`device.x`. Ele contém informação sobre os vetores de interrupção do MCU em questão. Esse arquivo tem o mesmo intuito que as funções `weak` em C, ele fornece uma ligação fraca que pode ser sobreposta em tempo de compilação. Sua aplicabilidade ficará mais clara na seção 4.4.3, com a discussão sobre a propriedades dinâmicas das aplicações.

4.4.2 Cortex-m

O pacote `cortex-m`, (RUST EMBEDDED, 2019b), foi criado para prover acesso de baixo nível aos periféricos do processador, com as respectivas APIs de consumo. Esse crate é a base do ecossistema Cortex-m® em Rust.

4.4.3 Cortex-m-rt

`Cortex-m-rt`, (RUST EMBEDDED, 2019c), é o crate responsável pela inicialização e pelo mínimo *runtime* do sistema. Contém todas as partes requeridas para construir um binário '`no_std`' que visa um microcontrolador Cortex-M®, descrito a seguir.

4.4.3.1 Inicialização

Em microcontroladores, sem sistema operacional, cabe à aplicação controlar toda a vida útil de execução, inclusive o processo de *boot*. No STM32L052 há três formas inicialização, configuráveis pelos pinos *BOOT0* e *BOOT1*. O modo de partida define qual região será apelidada para o início da seção CODE (Figura 4.1), variando entre a *flash*, memória do sistema, *SRAM* (STMICROELECTRONICS, 2017). Para o contexto desse trabalho não importa a região escolhida, nem o porquê da escolha. O fundamental é que após a energização do dispositivo, o início da seção CODE é ocupado pelos vetores de interrupção, considerando que o *linker* fez seu trabalho corretamente.

"Linkagem" é o processo final de compilação. Cabe ao *linker* produzir o executável através da combinação de diversos arquivos objeto. De forma simplificada, seu principal propósito é mapear as seções dos arquivos de entrada para o *layout* de memória desejado. Em Rust, o *linker* usado é o *LLVM-ld*, (LLVM PROJECT, 2019a). Seguindo a tendência de promover experiências ergonômicas de desenvolvimento, o *Cargo* fará com que esse processo seja transparente ao desenvolvedor da aplicação.

O arquivo `link.x.in`, (RUST EMBEDDED, 2019c), interno ao `cortex-m-rt`, é um *template* para um *linker script*. Os arquivos `device.x` e `memory.x`, vinculados ao MCU utilizado, preenchem as lacunas existentes. Ao executar um "`cargo run`" esses 3 arquivos se combinam gerando o `link.x`, o *linker script* final.

O arquivo `memory.x`, Listagem 4.9, informa o tamanho das porções de memória disponível no MCU. O pacote `cortex-m-rt` espera que ao menos duas regiões sejam informadas: `FLASH` e `RAM`. Por padrão, `.text` e `.rodata` serão colocadas na `FLASH`, e `.bss`, `.data` e `heap` ficarão na `RAM`. Esse arquivo norteará a seção `MEMORY` do *linker script*.

Listagem 4.9: Arquivo memory.x

```
MEMORY
{
  FLASH : ORIGIN = 0x08000000, LENGTH = 32K
  RAM : ORIGIN = 0x20000000, LENGTH = 8K
}
```

A utilização do arquivo `device.x` depende da ativação ou não da funcionalidade *device*. Quando desativada, o `cortex-m-rt` preencherá toda a tabela de interrupção com valores padrões, mesmo aquelas posições não usadas pelo MCU em questão. Nesse modo, a aplicação fica agnóstica ao dispositivo final, depende apenas da arquitetura. Quando *device* é habilitada, o pacote não se responsabiliza pelo preenchimento, cabendo à aplicação completá-la. O *crate* de *runtime*, através de um arquivo de construção especial, `build.rs` (Listagem 4.10), buscará o `device.x`. Por padrão todos os manipuladores, exceto *Reset* e *HardFault*, inicialmente apontam para o `DefaultHandler`, Código 4.11, representado por um *loop* infinito.

Listagem 4.10: O arquivo 'build.rs' deve criado para informar ao 'cargo' onde encontrar o arquivo 'device.x'.

```
use std::env;
use std::fs::File;
use std::io::Write;
use std::path::PathBuf;

fn main() {
    // Put the linker script somewhere the linker can find it
    let out = &PathBuf::from(env::var_os("OUT_DIR").unwrap());
    File::create(out.join("device.x"))
        .unwrap()
        .write_all(include_bytes!("device.x"))
        .unwrap();
    println!("cargo:rustc-link-search={}", out.display());
}
```

Figura 4.4: Parte da tabela de vetores de interrupção, extraído do manual referência da plataforma.

Position	Priority	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000_0000
	-3	fixed	Reset	Reset	0x0000_0004
	-2	fixed	NMI_Handler	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
	-1	fixed	HardFault_Handler	All class of fault	0x0000_000C
	-	-	-	Reserved	0x0000_0010 - 0x0000_002B
	3	settable	SVC_Handler	System service call via SWI instruction	0x0000_002C
	-	-	-	Reserved	0x0000_0030 - 0x0000_0037
	5	settable	PendSV_Handler	Pendable request for system service	0x0000_0038
	6	settable	SysTick_Handler	System tick timer	0x0000_003C

Fonte: (STMICROELECTRONICS, 2017)

Listagem 4.11: Manipulador padrão de interrupções e exceções. Código extraído da biblioteca 'cortex-m-rt'.

```
#[doc(hidden)]
#[no_mangle]
pub unsafe extern "C" fn DefaultHandler_() -> ! {
    loop {
        // add some side effect to prevent this from turning
        // into a UDF instruction
        // see rust-lang/rust#28728 for details
        atomic::compiler_fence(Ordering::SeqCst);
    }
}
```

Segundo descrito pelo manual de referência do dispositivo (STMICROELECTRONICS, 2017), após a energização, a CPU buscará o conteúdo da primeira posição de memória, 0x00000000, e o usará como *Main Stack Pointer* (MSP). No próximo passo atribui o conteúdo da segunda posição, 0x00000004, ao contador de programa, *Program Counter* (PC), definindo a próxima instrução a ser executada. A Figura 4.4, apresenta os vetores de interrupção, ratificando que a posição 0x00000004 é responsável pelo *Reset* do sistema.

Ao *Reset_Handler*, cabe, primeiramente, inicializar a memória RAM com suas devidas seções: *bss* (variáveis não inicializadas) e *data* (variáveis inicializadas). Após isso, ele configura a unidade de ponto flutuante em hardware, caso ela exista. E finalmente, o sistema está pronto para executar o código da aplicação.

Através dos *macros* `#[entry]` e `#[pre_init]`, o desenvolvedor da aplicação informa quais

serão os pontos de entrada da sua solução. O Código 4.12 ilustra o processo descrito acima. A definição completa da rotina de restabelecimento está no disponível no arquivo `lib.rs` em (RUST EMBEDDED, 2019c).

Listagem 4.12: Definição simplificada do processo de 'reset' do sistema.

```
1  #[no_mangle]
2  pub unsafe extern "C" fn Reset -> ! {
3      // Pre-inicializacao, definida pelo usuario,
4      // com o macro #[pre_init]
5      __pre_init();
6
7      // Inicializacao da RAM.
8      // O ‘crate‘ r0 ‘https://docs.rs/r0/0.2.2/r0/‘,
9      // abstrai a inicializacao do codigo.
10     r0::zero_bss();
11     r0::init_data();
12
13     #[cfg(has_fpu)]
14     enable_fpu();
15
16     // A funcao principal eh instanciada pelo usuario,
17     // via #[entry]
18     main()
19 }
```

A macro `#[entry]` garante que a função `main` é divergente, (RUST LANGUAGE, 2019a), isto é, que nunca finaliza. A ausência de sistema operacional em *background* implica no não término da aplicação. Em Rust, isso é assegurado durante a compilação.

4.4.3.2 *Runtime*

A *macro* `#[exception]` é estratégia idiomática para redefinir uma rotina de tratamento de exceção, isto é, sobrescrever o *DefaultHandler*. Sintaticamente, ele precede a função, *derive-like*, que será usada para remapear a exceção. O nome dessa rotina deve condizer com o esperado pelo *linker*. Marcar uma função qualquer como interrupção ou exceção implica em erro de compilação (linha 13, Listagem 4.13).

Por mais que o pacote `cortex-m-rt` implemente a *macro* `#[interrupt]`, ela não deve ser usado diretamente. Ao contrário das exceções que são comuns a uma arquitetura, as interrupções variam entre microcontroladores. Logo, essa funcionalidade deve ser importada do *crate* do dispositivo. O Código 4.13 exemplifica o uso das *macros* acimas.

Listagem 4.13: Como usar os macros 'interrupt' e 'exception' para sobrescrever o manipulador padrão.

```
#[exception]
fn SysTick() {
    ...
}

#[interrupt]
fn TIM2() {
    ...
}

// Erro de compilacao, pois a ‘funcao_qualquer‘ nao
// esta ligada a um vetor de interrupcao.
#[interrupt]
fn funcao_qualquer() {
    ...
}

// Erro de compilacao, interrupcoes e excecoes devem
// ser finitas e nao retornam nenhum valor.
#[exception]
fn funcao_divergente() -> ! {
    ...
    loop {}
}
```

Das configuração de *runtime,* falta apenas estabelecer o comportamento em caso de `panic!`. Enquanto nas aplicações tradicionais a biblioteca padrão se responsabiliza por isso, no mundo `no_std`, cabe à solução "explicar" o que acontecerá em caso de falha. Para tal, deve-se criar a função '`fn panic(_info: &PanicInfo) -> !`', precedida pelo decorador `#[panic_handler]`.

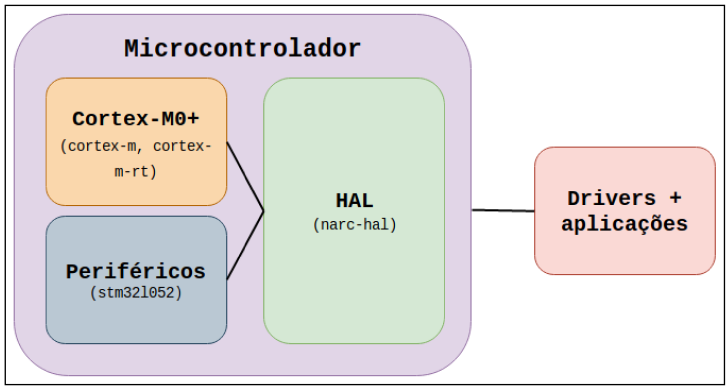
O artigo (LINDNER, M.; APARICIUS; LINDGREN, 2018) analisa as dificuldade em definir um procedimento de erro para aplicações embarcadas, ressaltando, mais uma vez, a busca por soluções estáticas de análise. Nos códigos mostradas nesse texto, o '`panic`' consistirá em um *break point* seguido por um *loop*, como mostrado na linha 35 da Listagem 4.16.

As configurações e sistemas mostrados acima foram criados considerando os casos mais comuns de uso. Se a aplicação em questão precisa ir além do convencionalmente decidido, tanto o processo de *startup* quanto a "linkagem" podem ser feitos de forma menos engessada. Mais detalhes estão disponíveis em (RUST EMBEDDED, 2019f), o *Embeddonomicon,* e na documentação dos próprios *crate*s.

4.4.4 Embedded-HAL

Com o acesso a nível de arquitetura provido pelos *crates* `cortex-m-rt` e `cortex-m` e dos periféricos do microcontrolador (*crate* `stm32l053`) gerado automaticamente, o próximo passo é a criação de abstrações de hardware. A organização desses `crates` no ecossistema *Embedded-Rust*

Figura 4.5: Organização básica dos *crates*.



Fonte: Autoria própria.

foi ilustrada na Figura 4.5.

A diferenças entre os dispositivos será superada com o uso do *Embedded-HAL*. Nesse pacote *traits* são propostos, discutidos, testados e disponibilizados, oferecendo contratos e interfaces comuns a microcontroladores e dispositivos externos (*drivers*) (RUST EMBEDDED, 2019d). Além de proporcionar a portabilidade entre dispositivos e reuso de códigos, ele provém a unificação de padrões reduzindo o emaranhado das aplicações.

Em Rust, construir o HAL (*Hardware Abstraction Layer*) de MCU significa implementar-lhe os *traits* prototipados no `Embedded-HAL`. Como contribuição, no presente trabalho é criado um HAL para o microcontrolador STM32L052 (SILVA, 2019b). Ele foi codificado adotando os preceitos definidos pela comunidade, sob influência de pacotes semelhantes.

A título de demonstração, o restante dessa seção apresentará a abstração empregada para construir uma interface serial com o periférico USART (*Universal Synchronous Asynchronous Receiver-Transmitter*). Note que ela consente com as ideias apresentadas na Seção 4.2 e, apoia-se na habilidade da linguagem em criar abstrações com custo zero.

4.4.4.1 UART

O protocolo universal de transmissão e recepção síncrono/assíncrono (*usart*) oferece uma maneira flexível de comunicação *full-duplex* em concordância com o padrão industrial NRZ (*non-return-to-zero*) (STMICROELECTRONICS, 2017). Com taxas de transmissão programáveis e comunicação *half-duplex* em fio único, ele é comumente aplicado em soluções industrias. No dispositivos ST,

o periférico homônimo implementa tal protocolo de hardware. Aqui, interessa apenas o modo assíncrono, isto é, sem clock compartilhado entre as partes.

É importante ressaltar que configurações equivocadas na UART podem gerar comportamentos indefinidos e seu uso incorreto expõe *data-races*. Como o contato com o mundo do físico se dá por meio de pinos de entrada/saída, a correta configuração desses será considerada na abstração da serial, visando aumentar a confiabilidade da solução.

A flexibilidade dos pinos de entrada/saída exposta na Figura 4.2 esbarra na utilização de funções alternativas. No que tange a elas, cada pino possui um número restrito de configurações possível. O desenvolvedor deve, consultando a ficha de dados do MCU, descobrir quais lhe atendem. Em uma abstração segura, somente os arranjos de pinos válidos devem ser permitidos. Com essa invariância tratada a nível design, mal uso dos pinos acarreta em erro de compilação.

Observe na Listagem 4.14 que, apenas os pares `(PA2<AF4>, PA3<AF4>)` (linha 22) e `(PA9<AF4>, PA10<AF4>)` (linha 25) implementam o comportamento esperado para os '`Pins`' da unidade 2 do periférico `USART`, somente eles serão aceitos para instanciar uma *serial-uart2*. Internamente, a '`Serial`' é genérica à qualquer conjunto de pinos que implemente o traço `Pins` (linhas 27, 30 e 37). Os outros argumentos da interface passam por avaliações semelhantes, por exemplo, o barramento precisa condizer com o esperado pelo manual (linha 33).

Listagem 4.14: Trechos de código da abstração serial - uart.

```
1  /// PA2 - USART2_tx PA3 - USART2_rx
2  impl Pins<USART2> for (PA2<AF4>, PA3<AF4>) {}
3
4  /// PA9 - USART2_tx PA10 - USART2_rx
5  impl Pins<USART1> for (PA9<AF4>, PA10<AF4>) {}
6
7  impl<PINS> Serial<PINS> {
8      pub fn usart2 (
9          usart: USART2,
10         pins: PINS,
11         baud_rate: Bps,
12         clocks: Clocks,
13         apb: &mut APB1,
14         character_match: Option<u8>,
15     ) -> Self
16    where
17       PINS: Pins<USART2>
18    {
19         apb.enr().modify(|_, w| w.usart2en().set_bit());
20         ...
21         usart
22             .cr1
23             .write(|w| w.ue().set_bit().re().set_bit().te().set_bit());
24
25         Serial { usart, pins }
26     }
```

```
27
28      pub fn split(self) -> (Tx<USART2>, Rx<USART2>)
29      { ... }
30  }
31
32  impl Read<u8> for Rx<USART2> {
33      type Error = Error;
34
35      fn read(&mut self) -> nb::Result<u8, Error> {
36          let sr = unsafe { (*USART2::ptr()).isr.read() };
37
38          if sr.pe().bit_is_set() {
39              nb::Error::Other(Error::Parity)
40          } else if sr.fe().bit_is_set() {
41              nb::Error::Other(Error::Framing)
42          } ...
43          else if sr.rxne().bit_is_set() {
44              // NOTE(read_volatile)
45              return Ok(unsafe {
46                  ptr::read_volatile(&(*$USARTX::ptr()).rdr
47                                     as *const _ as *const _)
48              });
49          } else {
50              nb::Error::WouldBlock
51          }
52      }
53  }
54
55  impl Tx<USART2> {
56      fn write(&mut self, byte: u8) -> nb::Result<(), !> {
57          // NOTE(unsafe) atomic read with no side effects
58          let sr = unsafe { (*$USARTX::ptr()).isr.read() };
59
60          if sr.txe().bit_is_set() {
61              unsafe {
62                  ptr::write_volatile(&(*$USARTX::ptr()).tdr
63                                      as *const _ as *mut _, byte)
64              }
65              Ok(())
66          } else {
67              Err(nb::Error::WouldBlock)
68          }
69      }
70  }
```

O consumo, empréstimo e variabilidade de cada parâmetro de entrada expressam, em alto nível, o comportamento interno do hardware. Após ser usada na 'Serial::usart2' (linha 8, Listagem 4.15), a estrutura USART2 não está mais disponível, garantindo que a variável 'serial' é o único ponto de acesso aos registros equivalentes. Já para habilitar o *clock* no periférico, a 'Serial' não deve "consumir" o 'APB1', assim, uma referência mutável é suficiênte (apb: &mut APB1 - linha 13).

Na Listagem 4.14 é possível também observar, na prática, o uso do *crate* stm32l052 (SILVA, 2018). O comando na linha 19 ativa o *bit* 'usart2en' do barramento APB1. A expressão na linha 23 escreve '1' nos *bits* ue (*USART enable*), re (*Receiver enable*) e te (*Transmitter enable*) do registro

`cr1` (*Control register* 1) da '`usart2`'.

O método `split` (linha 28) foi criado para separar a '`Serial`' em transmissor e receptor independentes. Para evitar *aliasing*, ele recebe o *ownership* da '`Serial`' e retorna '`Tx`' e '`Rx`'. Como envio/recepção de um byte não é imediato a nível de hardware, já que, envolve o deslocamento de bits entre registros, as funcionalidades de escrita e leitura retornam a variante de erro `WouldBlock` do *crate* `nb` (APARICIO, 2018b). Indicando, assim, que suas operações não tem efeito imediato e podem requerer uma operação de bloqueio para completar. Essa abstração, inspirada no `std::io::Result`, foi criada para ser agnóstica ao modelo de concorrência superior, permitido modelos como `futures`, `async/await` ou simplesmente espera ocupada (linha 17, Listagem 4.15).

O método `read` (linha 35) pode retornar o *byte* recebido (`OK(dr)`) ou uma indicação de erro. Caso o registro de status (`sr`) indique alguma falha de recebimento, tem-se um '`nb::Error::Other(Error::Parity)`', por exemplo. Se o "problema" for a ausência de dados a serem recebidos, informa-se ao usuário do HAL que, "receber um dado por implicar em bloquear a execução" ou `nb::Error::WouldBlock` (linha 50). Uma explicação semelhante conduz a criação do método `write`, se a escrita foi efetuada com sucesso, ele retorna `OK()` caso contrário informa que "`WouldBlock`".

Listagem 4.15: Uso da interface 'Serial'.

```
1  fn main() {
2      // Os pinos ‘tx‘ e ‘rx‘ precisam ser ‘PINS‘ aceitos.
3      let tx = gpioa.pa9.into_alternate(&mut gpioa.moder).af4(&mut gpioa.afrh);
4      let rx = gpioa.pa10.into_alternate(&mut gpioa.moder).af4(&mut gpioa.afrh);
5
6      // Instanciado a ‘Serial‘.
7      let serial = Serial::usart2(
8          hw.USART2,
9          (tx, rx),
10         9_600.bps(),
11         clocks,
12         &mut rcc.apb2,
13         None,
14     );
15
16     let (mut tx, mut rx, _) = serial.split();
17     block!(tx.write(b'X')).ok();
18     let data = block!(rx.read()).unwrap();
19 }
```

Nessa subseção apresentou-se como Rust possibilita que diversos conceitos de alto nível sejam incorporados ao design produzindo uma abstração confiável para o protocolo UART. Detalhes foram omitidos, como a configuração e utilização do DMA e, a leitura e recebimento de um vetor de bytes. A implementação completa está disponível em (SILVA, 2019b). De agora em diante, o

presente texto fará uso dos periféricos que lhe foram convenientes, desprezando possíveis detalhes construtivos. As implementações e documentações pertinentes à construção dos mesmos estão disponível no *crate* `narc_rust` (SILVA, 2019b).

4.4.5 Verificação e validação

A versão final do programa "acender um led" usando as ferramentas "oficiais" está disponível na Listagem 4.16. Na função `main()`, o *singleton* 'hw' coordena o acesso aos periféricos de placa; '`RCC.constrain()`' transfere à variável '`rcc`' o controle dos barramentos de *clock*; '`gpioa`' controla seu periférico homônimo; a variável '`led`' manipula o pino PA5 em modo de saída digital. Ao fim, '`led.set_high()`' coloca a respectiva saída em nível lógico alto.

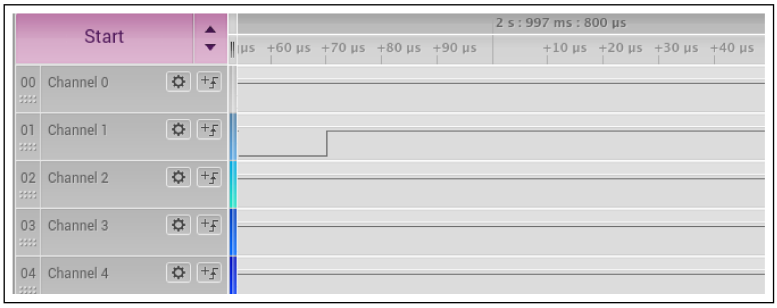
Listagem 4.16: Acendendo um led de forma idiomática em Rust.

```
#![no_std]
#![no_main]

use core::panic::PanicInfo;
use cortex_m::asm::bkpt;
use narc_hal::{stm32l052, rcc::RccExt, gpio::GpioExt};
use embedded_hal::digital::OutputPin;
use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    let hw = stm32l052::Peripherals::take().unwrap();
    let mut rcc = hw.RCC.constrain();
    let mut gpioa = hw.GPIOA.split(&mut rcc.iop);
    let mut led = gpioa.pa5
        .into_output(&mut gpioa.moder).push_pull(&mut gpioa.otyper);
    led.set_high();
    loop{
    }
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    bkpt();
    loop {
    }
}
```

Verificar um sistema é avaliar se ele cumpre os requisitos propostos. Nesse caso, se o Código 4.16, realmente acende um led. A Figura 4.6 mostra o sinal adquirido no pino PA5, quando tal código é executado. Note que condiz com o esperado, após o início da aplicação, a saída vai para nível lógico alto.

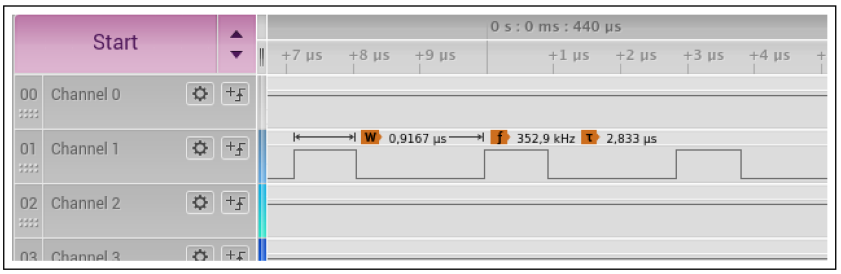
Já a validação tem um caráter mais amplo e deseja assegurar que as expectativas são atendidas. Aqui, é fundamental que a performance do código seja compatível com a realidade. A fim de rati-

Figura 4.6: Resposta no tempo do pino PA5 ao executar a Listagem 4.16. Sinal coletado por um analisador de sinais com taxa de aquisição de 12 MS/s.



Fonte: Autoria própria.

Figura 4.7: *Toggle* Led. Sinal coletado por um analisador de sinais com taxa de aquisição de 12 MS/s.



Fonte: Autoria própria.

ficar o Programa 4.16, o método `set_high` foi movido para dentro do `loop`, seguido pela instrução '`set_low`', como pode ser observado na Listagem 4.17. A Figura 4.7 apresenta o comportamento da aplicação no novo modo de operação, note a alternância no valor do pino de saída. O sinal apresentado nessa figura foi capturado através de uma analisador digital com taxa de aquisição de 12MS/s.

Listagem 4.17: Validação de um programa Rust embarcado. Loop com 'toggle' led.

```
1 loop{
2     led.set_high();
3     led.set_low();
4 }
```

Velocidade e tamanho de código são características contraditórias em otimizações computacionais. A melhoria de uma área pode ter impacto negativo na outra. Dado um trecho de código o compilador irá, provavelmente, tomar caminhos diferentes dependendo da priorização escolhida. O fino ajuste entre esse pontos vai além do escopo deste trabalho. As discussões abaixo analisam

a versão de *release* criada priorizando a velocidade da aplicação e com otimizações de linkagem habilitadas (*opt-level* = 3, *lto* = *true*).

Nesse modo, com *threshold* de otimização igual a 275, a LLVM não perderá oportunidades de otimizar o binário resultante (RUST EMBEDDED, 2019f), proliferando o uso de chamadas *inline* e outros artifícios de otimização. Conforme mostrado na Figura 4.7, a nível de aplicação, levou-se aproximadamente 1µs (2 ciclos) para sair do nível alto para o baixo e 4 ciclos para trocar de 0 para 1 (*clock* do sistema a 2097kHz MSI). Tal diferença decorre da implementação do `loop`, em baixo nível, o tempo de ciclos dos métodos `set_high` e `set_low` é o mesmo.

Com as devidas ponderações, o fragmento abaixo apresenta as seções de memórias (e seus respectivos tamanhos) da Listagem 4.17:

```
$ cargo size --example led --release -- -A
  Compiling narc_app v0.1.2
  Finished release [optimized] target(s) in 0.91s
led  :
section               size        addr
.vector_table          192   0x8000000
.text                  226   0x80000c0
.rodata                  0   0x80001a4
.data                    0  0x20000000
.bss                     4  0x20000000
```

Eficiência, em Rust, está intimamente ligada à exatidão da aplicação, não apenas à sua velocidade. O binário de *release* pode soar (des)otimizado, mas vale frisar que ele também é seguro. Isso pode parecer tolo, todavia nem sempre é verdadeiro para demais linguagens. Nelas, o comportamento do executável varia, profundamente, de acordo com o modo de compilação.

Vale ressaltar que essa seção não teve a pretensão de confrontar o binário Rust com aplicações C equivalentes, cujo binário final seria, provavelmente menor, mas sem as garantias da linguagem Rust. Algumas funcionalidades, para chegar ao canal *stable*, sofreram com perdas de performance (a princípio temporárias) afim de garantir a retrocompatibilidade. É o caso da instruções Assembly *inline* (RUST EMBEDDED, 2018). Alguns tópicos ainda estão sob discussão e os próximos anos devem dar maturidade ao ecossistema. Comparar as linguagens Rust e C iria requerer considerações profundas, fugindo ao escopo desse trabalho. O objetivo, aqui, foi apresentar a ordem de grandeza de uma aplicação Rust - com HAL - assegurando a viabilidade dos modelos propostos.

4.5 Considerações finais

Por limitação de escopo, nem todos os pontos associados à implantação da linguagem Rust em microcontroladores foram tratados. Tópicos chave, como o interfaceamento binário (e.g. alinhamento de dados e o "empacotamento" de estruturas) passaram despercebido. Rust se reserva o direito de não definir uma *Application Binary Interface* (ABI) estável, não há garantias de compatibilidade binária entre executáveis produzidos a partir de versões diferentes do compilador. Discussões acerca desse assunto são frequentes na comunidade (VARIOUS CONTRIBUTORS, 2015).

Como as aplicações exploradas por esse texto se comunicam diretamente com hardware, são necessários alguns cuidados especiais. As estruturas (ou `enums`), por exemplo, devem informar como desejam ser dispostas na memória. O mais usual é adotar as regras de alinhamento definidas pela linguagem C. Idiomaticamente basta anotar a estrutura de dados em questão com a indicação '`#[repr(C)]`'. Cabe salientar que o *crate* `sdv2rust` assegura que o 'periféricos' são C-compartíveis.

Os códigos apresentados até aqui se preocuparam apenas com o aspecto síncrono da aplicações embarcadas. Demonstraram como abstrações de configuração seguras se refletem em sistemas mais confiáveis. O próximo capítulo introduz a concorrência em aplicações microcontroladas, e como essa questão é tratada em Rust. Em seguida, apresenta a *framework* RTFM-Rust e os obstáculos encontrados ao criar uma aplicação nesse molde. Ao fim, expõe as extensões propostas por esse trabalho e como elas atuam nos pontos evidenciados.

Capítulo 5

Tempo Real e Concorrência em Rust Embarcado

A concorrência ocorre quando processos simultâneos competem por recursos em comum. Como os softwares embarcados reagem ao ambiente externo, a simultaneidade desponta mais cedo ou mais tarde. Em microcontroladores, esse problema se notabiliza nos manipuladores de interrupção. Procedimentos de interrupção e o fluxo principal, ao compartilharem informações, definem uma série de situações conflitantes. A próxima seção ilustra esses problemas, apresentando no fim uma versão simplificada do modelo de computação empregado pelo RTFM.

5.1 Modelos de concorrência

Quando a abordagem síncrona em super *loop* não é suficiente para cumprir os requisitos de tempo real, a concorrência entra em ação. Variáveis globais estaticamente alocadas são a escolha mais usual para o compartilhar informações entre os diferentes contextos da aplicação. As '`static mut` *variables'*, em Rust, indicam desde cedo seu comportamento inseguro, são '`unsafe`' para escrita e leitura. Usá-las exige cuidados especiais pois o compilador (*borrow checker*) não é capaz de assegurar, sozinho, a ausência de corridas de dados. Sua transformação para uma abstração segura passa pelo uso de algum mecanismos de sincronização[1].

[1] A abordagem usando células de mutabilidade interna (*Cell*) para evitar o uso de blocos *unsafe* explícitos foi omitida. Como essa abstração também não é "seguramente compartilhável" e requer os mesmos cuidados de uma variável global, apresentá-la detalhadamente levantaria questões que pouco acrescentam às discussões desse texto.

Em hardware, é possível atingir tal objetivo com o uso de instruções atômicas (`Atomic`), ou seja, operações indivisíveis. O mecanismo mais usual se apoia em "comparação e troca" - *compare-and-swap* (CAS), a fim de evitar que outros *threads*, erroneamente, se deparem com o estado intermediário de uma posição de memória. Essa funcionalidade está diretamente ligada à capacidade do microcontroladores, e não é encontrada em todas as plataformas. A arquitetura *ARMv6-M*, empregada no Cortex-M0+®, por exemplo, não dispõe de tais instruções (*ldrex* e *strex*). Vale ressaltar, que uma API 'atômica mínima', a partir barreiras de reordenamento (*fences*), está disponível para essas plataformas. Sua aplicação, entretanto, é restrita à situações mais específicas (VARIOUS CONTRIBUTORS, 2017).

Tais limitações, principalmente em termos de portabilidade, tornam as sincronizações a nível de software a escolha mais conveniente para a maioria aplicações embarcadas. Seções críticas evitam que interrupções aninhadas corrompam o valor da variável em questão. Em contrapartida, aumentam a complexidade da aplicação. A Listagem 5.1 é uma primeira (inocente) implementação do conceito descrito acima. O correto funcionamento desse sistema é de total responsabilidade do programador. Usos equivocados não implicam em erro aparente e podem se perdurar por muito tempo até que causem uma situação fora do esperado. A Listagem 5.1 mostra o uso da variável `COUNTER` quando essa está protegida por uma seção crítica, interrupções desabilitadas (`interrupt::free`, linha 5). Nesse design todo o controle da complexidade está com o programador.

Listagem 5.1: Seção crítica da variável 'COUNTER'.

```
1  static mut COUNTER: u32 = 0;
2
3  #[entry]
4  fn main() -> ! {
5      cortex_m::interrupt::free(|_| {
6          unsafe { COUNTER += 1 };
7      });
8  }
9
10 #[interrupt]
11 fn timer() {
12     unsafe { COUNTER = 0; }
13 }
```

Protocolos de compartilhamento de acesso, como a exclusão mútua, vieram para aprimorar tais abstrações. *Mutex* são algoritmos (as vezes estruturas de dados) usados para proteger regiões livres de interrupção. Quando usados inadequadamente, pode resultar em *deadlocks* e inversões de

prioridade (perdas de deadline). Na maioria das linguagens de programação facilmente acarretam em equívocos de implementação por requererem aquisição e liberação explícita. Em Rust, os *mutex* foram projetados para se encaixar dentro do sistema de *ownership*. Por design, estão ligado a um *'token'* de uma seção crítica. Ao final do escopo da variável protegida, ele é automaticamente liberado, como demonstrado na Listagem 5.2.

Listagem 5.2: Exemplificação do comportamento de um Mutex em 'Rust embarcado'. Ao final da seção crítica 'cs' há a liberação de 'COUNTER'.

```
use core::cell::Cell;
use cortex_m::interrupt::Mutex;

// ‘Cell‘ fazem parte do modelo de mutabilidade interna.
// Provem um sintaxe mais conveniente ao trabalhar com
// variaveis compartilhadas mutaveis. Deslocam as operacoes
// ‘unsafe‘ para uma API segura, validaveis pelo compilador
// em tempo de execucao. Com isso, possuem custos de execucao.
// Mais informacoes:
// https://doc.rust-lang.org/std/cell/
static COUNTER: Mutex<Cell<u32>> = Mutex::new(Cell::new(0));

#[entry]
fn main() -> ! {
    ...;
    interrupt::free(|cs|
                    COUNTER.borrow(cs).set(COUNTER.borrow(cs).get() + 1));
}

#[interrupt]
fn timer() {
    interrupt::free(|cs| COUNTER.borrow(cs).set(0));
}
```

O design apresentado em 5.2 desfruta de algumas conveniência, todavia, não elimina os problemas de alto nível. Além disso, em seu estado atual presume o uso exagerado de seções críticas. Em determinadas situações é seguro manipular uma variável compartilhada sem que seja necessário desativar as interrupções do sistema. Fundamentado no modelo descrito na Seção 2.3, o escalonador de tarefas empregado pela plataforma RTFM oferece garantias estáticas de que as propriedades concorrentes são seguramente atendida. Ao mesmo tempo oferecem a implementação mais otimizada possível em baixo nível.

Através de análises estáticas, o *framework* RTFM transforma um modelo simplificado (*syntactic sugar*) em procedimentos eficientes. Onde necessário, assegura o acesso do recurso via *mutex*, nas demais situações apenas "esconde" a insegurança das variáveis globais. O restante de capítulo se preocupa com as concepções práticas relacionadas à plataforma e como elas se refletem na construção de aplicações RTFM-Rust.

5.2 RTFM

Originalmente desenvolvida em OCaml, a *framework* está sendo portada para Rust por Jorge Aparicio, (APARICIO, 2019f). Aqui, RTFM-*lang* é implementada como um linguagem de domínio específico, através de macros procedurais. Ao presente tempo, a versão estável da plataforma é a V0.4.2, sendo essa a ser considerada no decorrer do texto. O RTFM parte do pressuposto de que o hardware no qual o programa será executado suporta despacho preemptivo e sobreposto. A estrutura básica de um programa pode ser visto na Listagem 5.3, as funcionalidades relevantes estão internas à *macro* `APP`.

5.2.1 DSLs em Sistemas de Tempo Real

A criação de DSLs para sistemas embarcados de tempo real não é novidade. Autores como (HAMMOND; MICHAELSON, 2003) defendem que a utilização de linguagens de propósito geral, *General Purpose Languages* (GPL), em tal domínio, para muitos casos, leva a um fraco ajuste entre os recursos da linguagem e as especificações de implementação. Já as linguagens de aplicação específica permitem que os requisitos de baixo nível guiem as características da linguagem de alto nível. Oferecem ganhos substanciais em expressividade (MITCHELL, 1993) e facilidade de uso (MERNIK; HEERING; SLOANE, 2005), reduzindo a complexidade acidental.

O desenvolvimento de uma DSL é um trabalho árduo, e exige conhecimento do problema e habilidade em desenvolver linguagens (MERNIK; HEERING; SLOANE, 2005). Enquanto as GPL vieram para aumentar a produtividade sobre Assembly, DSL buscam o mesmo efeito sobre linguagem de aplicações gerais (MERNIK; HEERING; SLOANE, 2005).

5.3 Desafios e concepções de uma aplicação RTFM-Rust

Em dispositivos Cortex-M®, o controle dos serviços de interrupção é responsabilidade do NVIC. Esse periférico, independente do processador, coordena a atribuição de prioridade ao manipuladores e inicialização dos mesmos a medida em que os eventos chegam. Em caso de interrupção, o NVIC decide qual função será executada em seguida e realiza a troca de contexto, liberando o processador para executar o trabalho realmente necessário. Como o RTFM se aproveita do con-

trolados de interrupção, a sobrecarga de escalonamento é zero.

Listagem 5.3: Estrutura básica de uma aplicação usando o RFTM-Rust.

```
1  #![no_main]
2  #![no_std]
3
4  use rtfm::app;
5  use rtfm::export::wfi;
6  use core::panic::PanicInfo;
7
8  use cortex_m::asm::bkpt;
9
10 #[app(device = narc_hal::stm32l052)]
11 const APP: () = {
12     static mut EXCLUSIVE: bool = false;
13     static mut SHARED: bool = false;
14
15     #[init]
16     fn init() {
17     }
18
19     #[idle(resources = [SHARED, EXCLUSIVE])]
20     fn idle() -> ! {
21         loop {
22             wfi();
23             *resources.EXCLUSIVE = true;
24
25             resources.SHARED.lock(|shared| *shared = false);
26         }
27     }
28
29     #[exception(resources = [SHARED], priority = 4)]
30     fn SysTick() {
31         *resources.SHARED = true;
32     }
33 };
34
35 #[panic_handler]
36 fn panic(_info: &PanicInfo) -> ! {
37     bkpt();
38
39     loop {
40     }
41 }
```

As tarefas são a unidade de concorrência do RTFM, elas podem ser *hard tasks* ou *soft tasks*. As tarefas de hardware são diretamente ligadas a um mecanismos de tratamento de interrupção. Caso seja uma exceção, a função é decorada pelo identificador `#[exception]`. Se for uma interrupção, ela será precedida por `#[interrupt]`. O nome da função deve ser o esperado pelo *crate* `stm32l052`. Decorar procedimentos que não estão ligados ao NVIC, como interrupção ou exceção, resulta em erro de compilação (linha 34, Listagem 5.4). Há uma RFC aprovada para que, na versão 0.5.x, essa ligação possa acontecer de forma mais facilitada (*syntactic sugar*), através do atributo *binds*, (APARICIO, 2019a). As tarefas de software serão exploradas na Subseção 5.4.

RTFM-*lang* possui duas tarefas especiais, `init` e `idle`. A função `init` é a primeira a executar,

quando o sistema é inicializado. Ela roda sob seção crítica geral e é usada para configurar os periféricos e instanciar os "recursos tardios" (*late resources*). Já função `idle` rodará em *background* quando nenhuma outra tarefa estiver disponível para escalonamento, ou melhor, enquanto nenhuma outra tarefa estiver pronta para executar. Ela tem a menor prioridade do sistema, 0 e é uma função divergente. Caso seu conteúdo ou protótipo não sigam essa regra tem-se um erro de compilação (linha 7, Listagem 5.4).

Listagem 5.4: Principais erros capturados durante a compilação de um aplicação RTFM.

```
1  #[app(device = narc_hal::stm32l052)]
2  const APP: () = {
3      static mut EXCLUSIVE: bool = false;
4      static mut SHARED: bool = false;
5
6      // error: ‘idle‘ must have type signature ‘[unsafe] fn() -> !‘
7      #[idle]
8      fn idle() {
9          loop {
10             wfi();
11             // error[E0614]: type ‘resources::SHARED<’_>‘ cannot be dereferenced
12             // O recurso SHARED eh requerido em duas funcoes.
13             // Na tarefa de menor prioridade seu acesso deve ser
14             // protegido por lock.
15             *resources.SHARED = false;
16
17
18             // error[E0593]: closure is expected to take 1 argument,
19             // but it takes 2 arguments
20             resources.SHARED.lock(|shared, shared2|
21                                   *shared = false);
22         }
23     }
24
25     #[exception(resources = [SHARED])]
26     fn SysTick() {
27         // error[E0609]: no field ‘EXCLUSIVE‘ on type ‘__rtfm_internal_8<’_>‘
28               // EXCLUSIVE nao esta na lista de recursos por essa tarefa.
29               *resources.EXCLUSIVE = true;
30     }
31
32     // error: only exceptions with configurable priority
33     // can be used as hardware tasks
34     #[exception]
35     fn test_exception() {
36     }
37 };
```

Com exceção dessas tarefas especiais, todas as outras, por padrão, possuem prioridade 1. Para alterar o nível de prioridade de uma tarefa é necessário adicionar o atributo `priority` em sua decoração, conforme a Listagem 5.3 linha 29. O menor nível (0) é conhecido para todas as arquiteturas, já a maior prioridade depende do microcontrolados usado. O valor `NVIC_bits` é herdado do *crate* `cortex-m` e norteia a priorização em hardware. Valores acima do aceito pelo MCU oca-

sionam falhas em tempo de execução, via `assert!(#priority <= (1 « #nvic_prio_bits))`.

Os itens '`(static mut ...)`' declaram os recursos protegidos pelo protocolo SRP, eles são estáticos para manter o valor entre chamadas. O primeiro recurso `EXCLUSIVE` é requerido apenas pela função `idle`, através da expressão `#[...(resources = [.., EXCLUSIVE, ..])]`, em seu decorador. Como não é feito compartilhamento, nenhum cuidado especial é necessário. Ele pode ser usado normalmente, através da expressão: `resources.EXCLUSIVE`. Acesso a um recurso fora do contexto da tarefa resulta em erro de compilação, vide a tarefa *SysTick* (linha 29) na Listagem 5.4.

O recurso `SHARED` é compartilhado entre a função `idle` e a rotina de tratamento do *SysTick*. Para acessá-lo de forma segura, na tarefa "menos prioritária", é necessário criar uma seção crítica (linha 15, Listagem 5.3). Pela sintaxe da DSL, isso é feito através de um `lock`. Ele cria um escopo interrompido e executa a função de primeira classe recebida. A maneira mais fácil e idiomática de transmitir uma função, em rust, é através de uma closure, algo como `|shared| *shared = false`. A rotina interrompida recebe um argumento, o recurso, e opera sobre ele. Procedimentos com qualquer outra quantidade de parâmetros resultam em erro de compilação, vide linha 20 da Listagem 5.4. Espera-se uma função que implemente o *trait* `FnOnce(&mut T) -> R`, ou seja, função que recebe `T` e retorna `R` (RUST LANGUAGE, 2019f).

5.3.1 Implementação

Ao reivindicar uma seção crítica, o teto do sistema é comparado ao teto do recurso em questão. Em dispositivos que suportam o definição de um nível mínimo de prioridade, como o registro ARM® - BASEBRI (ARM, 2012), a aplicação do protocolo é direta. Se $[R] < \Pi$, o valor do BASEPRI é mantido e a seção crítica é executada. Caso contrário, o BASEPRI é atualizado para o teto do recurso, a seção crítica será executada, ao final retorna-se ao Π valor anterior (Listagem 5.5). Para arquiteturas que não possuem tal funcionalidade, há duas alternativas. Originalmente, previa-se a criação de máscaras para desabilitar apenas as interrupções pertinentes (LINDGREN; LINDNER, M. et al., 2015). Entretanto, em Rust, caso o $[R]$ seja maior do que a prioridade da função corrente todas as interrupções são desabilitadas (APARICIO, 2018a), como na Listagem 5.6.

Listagem 5.5: Reivindicação de um recurso em uma plataforma com BASEPRI.

```
1 #[cfg(armv7m)]
```

```
#[inline(always)]
pub unsafe fn claim<T, R, F>(
    ptr: *mut T,
    priority: &Priority,
    ceiling: u8,
    nvic_prio_bits: u8,
    f: F,
) -> R
where
    F: FnOnce(&mut T) -> R,
{
    let current = priority.get();

    if priority.get() < ceiling {
        if ceiling == (1 << nvic_prio_bits) {
            priority.set(u8::MAX);
            let r = interrupt::free(|_| f(&mut *ptr));
            priority.set(current);
            r
        } else {
            priority.set(ceiling);
            basepri::write(logical2hw(ceiling, nvic_prio_bits));
            let r = f(&mut *ptr);
            basepri::write(logical2hw(current, nvic_prio_bits));
            priority.set(current);
            r
        }
    } else {
        f(&mut *ptr)
    }
}
```

Listagem 5.6: Seção crítica em uma arquitetura sem BASEPRI. Todas as interrupções são desabilitadas.

```
#[cfg(not(armv7m))]
#[inline(always)]
pub unsafe fn claim<T, R, F>(
    ptr: *mut T,
    priority: &Priority,
    ceiling: u8,
    _nvic_prio_bits: u8,
    f: F,
) -> R
where
    F: FnOnce(&mut T) -> R,
{
    let current = priority.get();

    if priority.get() < ceiling {
        priority.set(u8::MAX);
        let r = interrupt::free(|_| f(&mut *ptr));
        priority.set(current);
        r
    } else {
        f(&mut *ptr)
    }
}
```

É importante notar que o tratamento equivocado dos recursos resulta em erro de compilação. Um código com *data-races* não compila, fundamental para a confiabilidade das aplicações. A por-

tabilidade para Rust está em pleno desenvolvimento e nem todos as funcionalidades previstas na versão original foram traduzidas. Concomitante a esse texto, diversas discussões sobre melhorias sintáticas e de performance estão acontecendo (APARICIO, 2019b,h).

5.3.2 WCET

Estimar o pior tempo de execução, *Worst-case execution time* (WCET), é crucial no desenvolvimento de software de tempo real. Ele assegura que o programa não irá exceder o tempo permitido. Métodos de análise da resposta no tempo e da escalabilidade são, normalmente, dependentes do conhecimento prévio do WCET. A verificabilidade de um sistema baseado no SRP, depende do conhecimento do pior caso de execução e do o tempo de retenção dos recursos (LINDNER, M.; APARICIO et al., 2018). O WCET pode ser estimado através de análises estáticas ou dinâmicas.

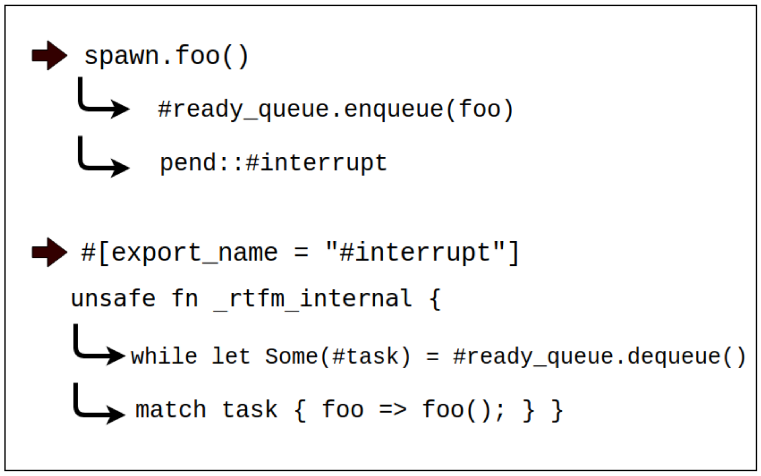
Ao escolher pelo processo estático abre-se mão da ligação como hardware, desprezando *cachés*, *pipelines* e outras características particulares à plataforma, com a vantagem de poder ser feito no computador de desenvolvimento.

Os métodos medidos em hardware são o padrão *de facto* (LINDNER, M.; APARICIO et al., 2018). Como benefício, não demandam tanto conhecimento teórico e são livres de suposições e aproximações associadas aos modelos. Entretanto, normalmente requerem procedimentos manuais suscetíveis a erros (LINDNER, M.; APARICIO et al., 2018). Em Rust, aproveitando-se da ferramenta KLEE (CADAR; DUNBAR; ENGLER, 2008), para a geração automatizada de vetores de teste, no artigo (LINDNER, M.; APARICIO et al., 2018) é apresentada uma forma completamente automática para mensurar o WCET em aplicações Rust-RTFM.

5.3.3 *Stack Overflow*

Além de ser uma política de acesso aos recursos, o SRP, é uma forma eficiente de controlar e mensurar a memória (BAKER, 1990). Para sistemas embarcados críticos, é fundamental saber se a aplicação sofrerá com corrupções de memória devido a estouro de pilha (*stack overflow*). É, aliás, um requisitos para certificação e confiabilidade funcional. O ideal é que a análise, do pior caso de consumo de stack, ocorra em tempo de compilação. Ferramentas para mensurar o consumo de pilha em C/C++ são, normalmente, onerosas.

Figura 5.1: Fluxo de despacho (simplificado) de uma tarefa de software.



Fonte: Autoria própria

O projeto `cargo-call-stack` (APARICIO, 2019g) pretende criar uma ferramenta para mensurar o pior caso de consumo de pilha para aplicações embarcadas escritas em Rust. Disponibilizado como um software aberto, funciona, de forma simplificada, segundo três processos. São eles: primeiro, calcula-se o *stack* de cada tarefa individualmente, depois percorre-se o grafo das chamadas do sistema. Ao fim, combinam-se as duas informações. A versão 0.1.2 ainda não suporta integração direta ao RTFM (APARICIO, 2019d), porém está no *backlog* de desenvolvimento do projeto fornecê-la em breve.

5.4 Tarefas de software

As tarefas de hardware são criadas para tratar eventos diretamente ligados a uma interrupção hardware, sendo cada uma mapeada a um manipulador único do NVIC (*Interrupt Service Routine*). Já as tarefas de software são muito úteis quando deseja-se manter o hardware responsivo, postergando ações menos críticas. Ela podem ser disparadas de acordo com a demanda do sistema. São priorizáveis e seguem as mesmas regras de preempção das tarefas de hardware. Para criá-las é necessário preceder a função correspondente com o atributo `#[task]`. Internamente, o escalonamento das tarefas de software também se apoiam controlador de interrupção.

Tarefas de software com a mesma prioridade compartilham uma fila, ligada a apenas uma posição do NVIC. De forma simplificada, o processo de despacho de uma tarefa de software pode ser

descrito segundo a Figura 5.1. Para cada nível de prioridade com *soft tasks* é necessário disponibilizar um *interrupt_handler*, através do bloco `extern "C"{}` (linha 46, Listagem 5.7). Semelhante às *hard tasks,* as tarefas de software são invocadas assíncronamente e não possuem valor de retorno.

Listagem 5.7: Aplicação RTFM com tarefas de software.

```
#![no_main]
#![no_std]

use rtfm::app;
use core::panic::PanicInfo;

use cortex_m::asm::bkpt;

#[app(device = narc_hal::stm32l052)]
const APP: () = {
    #[init]
    fn init() {
    }

    #[idle(spawn = [tarefa_software])]
    fn idle() -> ! {
        loop {
            let result = spawn.tarefa_software(true);
            // O `spawn` de uma tarefa retorna um Result.
            // Que pode ser tratado de duas maneiras:
            // via `match`
            match result {
                Ok(_) => (), //tarefa agendada com sucesso.
                Err(_) => (), //erro ao agendar a tarefa.
            };
            // Ou, `panic!` em caso de erro.
            result.unwrap();

            // error[E0609]: no field `outra_tarefa` on type `idle::Spawn<'_>`
            spawn.outra_tarefa.unwrap();

            tarefa_sincrona(...);
        }
    }

    #[task(priority = 3)]
    fn tarefa_software(_value: bool) {
    }

    // error: 2 free interrupts (`extern { .. }`) are required
    // to dispatch all soft tasks
    #[task]
    fn outra_tarefa(_value: bool) {
    }

    extern "C" {
        fn TIM2 ();
    }
};

// Note, a funcao_sincrona eh externa ao macro APP.
fn tarefa_sincrona(shared: impl Resource<...>) {
    shared.lock(|shared| {
        ...
    });
}
```

Pela sintaxe atual, as funções síncronas são procedimentos comuns, utilizam os mecanismos comuns de função em Rust, como a `funcao_sincrona` (linha 52) na Listagem 5.7 e são posicionadas fora do macro `APP`. Herdam a prioridade da função mãe. Portanto, em determinados casos, o acesso aos recursos deve ser feito via `RTFM::mutex` (linha 53, Listagem 5.7). Há uma discussão sobre uniformização de sintaxe entre tarefas síncronas e assíncronas, RFC (APARICIO, 2019e). Ao presente tempo, não há consenso e a decisão foi postergada para versões futuras.

O agendamento de um tarefa de software é feito através do comando `spawn`. Para ter a capacidade de liberar uma *soft task*, a tarefa mãe precisa ter essa informação declarada em seus atributos, `#[...(spawn = [..., tarefa_software, ...])]` . Apenas tarefas explicitamente colocadas no contexto, podem ser escalonadas. Um erro de compilação é disparado caso tente-se usar uma tarefa não pertinente ao contexto (linha 30, Listagem 5.7).

Cada um dos manipuladores de interrupção de *soft-task* possuirá uma lista de "tarefas prontas" (*ready queue*). A expressão `spawn.<tarefa>()` adiciona a `<tarefa>` à lista de tarefas a executar e ativa o bit da interrupção correspondente no NVIC com a função `pend(<interrupt>)` disponível em (RUST EMBEDDED, 2019c). No manipulador de interrupção há o processo contrário, as tarefas prontas são desenfileiradas e as funções correspondentes são chamadas.

É possível enviar informações a tarefas de software, sem compartilhamento explícito de memória. Na Listagem 5.7, a função `idle` envia o valor *"true"* para a `tarefa_software`. Além disso, ao contrário das tarefas de hardware, as *soft tasks* podem ser "`spawn`*ed"* múltiplas vezes. As informações (mensagens) serão enfileiradas e disparadas na ordem em que entraram, primeira a entrar - primeira a sair, *First In First Out* (FIFO). Todos os *buffers* internos são alocados estaticamente, caso sua capacidade "estoure", tem-se um erro e a tarefa não é escalonada (linhas 18-27). A menos que explicitado via `capacity`, a capacidade padrão de uma tarefa é 1.

5.4.1 Software *timers*

A habilidade de escalonar eventos futuros depende do domínio de serviços de tempo. *Timers* são periféricos similares a um alarme. Após o intervalo de tempo configurado, eles emitem um evento. Sistemas complexos são compostos por diversos componentes e funcionalidades, requerindo diversos valores de tempo limite. Para contornar a limitação de *timers* disponíveis, a maioria *dos firmwares* dividem as atividades de tempo em dois grupos: *hard timers*, *soft timers* (LI; YAO, 2003).

Hard timers estão ligados diretamente a um periférico. Operações com requisitos exigentes de precisão ou latência precisam do previsível desempenho de um *hard-timer*. Os *soft-timers* são eventos de software agendados por meio de um recurso de software (LI; YAO, 2003). No RTFM, a temporização por software é efetuada através de filas de tempo (*timer queues* - TQ). O artigo (LINDGREN; FRESK et al., 2016), apresenta a abordagem adotada e como o hardware subjacente é controlado.

A precisão e faixa de operação de uma fila de temporizadores é dependente de características da unidade física ao qual ela está ligada. A exatidão (pr) pode ser calculada segundo a fórmula $pr = 1/f$, onde f é a frequência de operação. Já a faixa de atuação em segundos (ra) é vinculada à largura de bits (bw) e a frequência de operação, segundo a fórmula: $ra = 2^{bw}/f$. (LINDGREN; FRESK et al., 2016) apresenta as duas estratégias de implementação de TQ, disponíveis na versão original da *framework*: via *SysTick* + DWT (*Debug Unit Timer*), ou através de *timers* genéricos do microcontrolados (múltiplas filas).

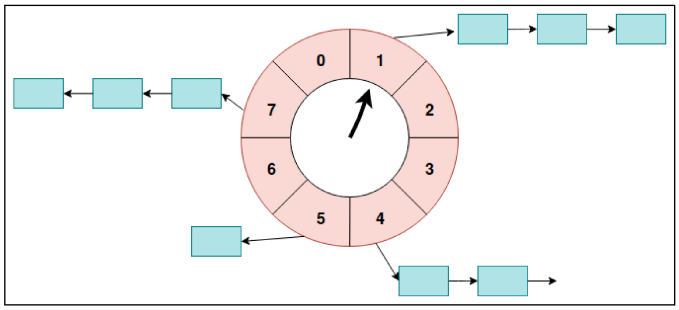
Em Rust, apenas a implementação utilizando o timer interno ao processador e a contagem de ciclos da unidade de depuração, *SysTick* + DWT - fila única, está disponível. Restringindo o uso à plataformas ARM7-M®. Sintaticamente, escalonar uma tarefa a uma fila de tempo é semelhante a disparar uma *soft-task* tradicional. Ao invés de `"spawn"` a tarefa será `"schedule"` com o devido tempo de atraso, por exemplo `schedule.foo(10.cycles())`. Após dez ciclos a tarefa `foo` será escalonada para execução. Devido à falta de suporte à plataforma usada nesse trabalho, detalhes de funcionamento e características de herança de tempo serão omitidos deste texto.

5.4.1.1 *Timer Wheel*

Para contornar a falta de um *soft-timer*, o presente trabalho, apresenta como contribuição a implementação de uma roda "temporizada" (*timer wheel* - TW) simples para o agendamento de tarefas, *via crate* `timer_wheel` (SILVA, 2019b). Na Figura 5.2 é ilustrado o funcionamento básico de uma TW genérica. A cada *tick* da roda, os eventos associados à posição em questão são disparados. A abstração não é completamente transparente ao usuário, permitindo sua utilização em mais do que um cenário. Para cada posição equivaler a um pulso do *timer* físico a função `tick` deve estar no manipulador de interrupção, como na Listagem 5.8.

Listagem 5.8: Cada pulso do timer 6 equivale a um 'tick' da timer-wheel.

Figura 5.2: Esquemático de funcionamento de uma roda temporizada.



Fonte: Autoria própria.

```
#[app(device = narc_hal::stm32l052)]
const APP: () = {
    #[interrupt(resources = [TW], spawn = [callee])]
    fn TIM6_DAC() {
        let to_run = resources.TW.tick();

        while let Some(func) = to_run.pop() {
            match func {
                Functions::Callee => spawn.callee().unwrap(),
            }
        };
    }
};
```

Seguindo a tendência determinística das aplicações em tempo real, a *timer-wheel* implementada não usa estruturas de dados dinâmicas. O *buffer* circular é emulado por um vetor *heapless* (Subseção 5.4.2.1). Em cada posição da roda há um outro vetor estático. A roda aceita eventos enquanto existirem posições válidas, após isso, um 'Err' é retornado.

Os tamanhos são fixos em tempo de compilação para evitar fragmentações e realocações de memória. Graças às estruturas estáticas, o tempo de inserção e remoção de elementos da TW é constante. Exemplos de utilização estão disponíveis em (SILVA, 2019a), no módulo de `tests` do pacote.

5.4.2 Coleções

O uso de estrutura de dados dinâmica, em sistemas de tempo real, não é recomendado. Sua imprevisibilidade dificulta a análise formal do programa (RIVERA; LINDNER, M.; LINDGREN, 2018). O *crate* `heapless`, (APARICIO, 2019c), provém em uma série de estruturas estática para programas em Rust. Apoiado no armazenamento de memória com capacidade definida em tempo de compi-

lação, ele foi construído agnosticamente ao alocador de memória e pode ser usado no *stack* ou no *heap*. A natureza estática implica que não haverá realocação de memória, resultando em tempos de operação realmente constantes (RIVERA; LINDNER, M.; LINDGREN, 2018), fundamental para atender os requisitos temporais dos problemas tratados.

Como as estruturas `heapless` não usam alocação dinâmica, não há estouro de memória (*Out Of Memory*) em tempo de execução. O *crate* busca oferecer uma experiência semelhante à encontrada nas estruturas da biblioteca padrão, compartilhando, onde possível a mesma API. A biblioteca oferece um série de estruturas de dados, são elas:

- *BinaryHeap*
- *IndexMap*
- *IndexSet*
- *LinearMap*
- *Single Producer Single Consumer(SPSC) - Queue*
- *String*
- *Vec*
- *Memory pool*

5.4.2.1 Vec

`Vec` é um vetor com capacidade fixa, definida em tempo de compilação. O código 5.9, demonstra suas principais operações. Enquanto o `push` da biblioteca padrão pode implicar em realocação, o `push` `heapless` retorna um `Result<(), T>`. Ao tentar tentar exceder a capacidade previamente definida, a variante `Err(T)` é retornada, indicando que o elemento não foi inserido na coleção. As garantias de segurança são mantidas, acesso fora dos limites do vetor resulta em `panic!`.

Listagem 5.9: Exemplo de utilização de um vetor ‘heapless‘.

```
1 use heapless::Vec;
2 use heapless::consts::*;
3
4 // Vetor, com a capacidade fixa de 8 elementos,
```

```
// alocado em ‘stack‘.
let mut vec = Vec::<_, U8>::new();
vec.push(1);
vec.push(2);

assert_eq!(vec.len(), 2);
assert_eq!(vec[0], 1);

assert_eq!(vec.pop(), Some(2));
assert_eq!(vec.len(), 1);

vec[0] = 7;
assert_eq!(vec[0], 7);

vec.extend([1, 2, 3].iter().cloned());

for x in &vec {
    println!("{}", x);
}
assert_eq!(vec, [7, 1, 2, 3]);

// A operacao abaixo resultara em ‘panic!‘
let _ = vec[10];
```

5.4.2.2 Single Producer Single Consumer - Queue

A `spsc::Queue` é uma fila, alocada estaticamente, com uma capacidade de N elementos. Como outra fila qualquer, expõe uma API de enfileiramento (`enqueue`) e uma de desenfileiramento (`dequeue`), vide 5.10.

Listagem 5.10: Exemplo de uma fila ‘heapless‘. API padrão ‘enqueue‘ e ‘dequeue‘.

```
use heapless::spsc::Queue;
use heapless::consts::*;

let mut rb: Queue<u8, U2> = Queue::new();

assert!(rb.enqueue(0).is_ok());
assert!(rb.enqueue(1).is_ok());
assert!(rb.enqueue(2).is_err()); // full

assert_eq!(rb.dequeue(), Some(0));
```

Geralmente, algoritmos e estruturas para sistemas concorrentes se enquadram em duas categórias: *blocking* e *non-blocking*. Estruturas com bloqueio permitem atrasos ou congelamento de processos para impedir acessos concorrentes em uma estrutura compartilhada (MICHAEL; SCOTT, 1995), com mutex e semáforos. Já os algoritmos *non-blocking* garantem que se um ou mais processos estão tentando realizar operações em um recurso compartilhado, essa operação terminará em um número finito de passos (MICHAEL; SCOTT, 1995).

Na programação assíncrona, os algoritmos bloqueadores (como *mutexes*) podem reduzir o de-

sempenho do sistema, incluindo perdas de preempção (inversão de prioridade). Algoritmos não bloqueadores são mais robustos a esses problemas (MICHAEL; SCOTT, 1995), sendo a escolha recomendada quando possível. Algoritmos *lock-free* são *non-blocking*.

O poder da `spsc::Queue` é amplificado quando ela é separada (`split`) em um único produtor e um único consumidor (Listagem 5.11). As APIs de consumo e produção são *lock-free*, ou seja, diferentes contextos de execução podem usar cada um dos *end point*, sem que exista uma trava global.

Listagem 5.11: Fila lock-less único produtor único consumidor 'heapless'.

```
static mut RB: Option<Queue<Event, U4>>  = None;

enum Event { A, B }

fn main() {
    unsafe { RB = Some(Queue::new()) };
    // NOTE(unsafe) notificando a existencia de aliasing no 'endpoint
    let mut consumer = unsafe { RB.as_mut().unwrap().split().1 };

    loop {
        // Note que o desenfileiramento nao envolve um 'lock' global.
        match consumer.dequeue() {
            Some(Event::A) => { /* .. */ },
            Some(Event::B) => { /* .. */ },
            None => { /* sleep */},
        }
    }
}

// Manipulador de interrupcao quaisquer,
// executa em contexto diferente e
// pode preempcionar a main.
fn interrupt_handler() {
    // NOTE(unsafe) notificando a existencia de aliasing no 'endpoint'.
    let mut producer = unsafe { RB.as_mut().unwrap().split().0 };

    // Note que o enfileiramento nao envolve um 'lock' global.
    if condition {
        producer.enqueue(Event::A).ok().unwrap();
    } else {
        producer.enqueue(Event::B).ok().unwrap();
    }
}
```

O comportamento livre de '`lock`' é implementado através de leituras e escritas atômicas nos métodos `enqueue` e `dequeue`. Em (LAMPORT, 1983), Lamport prova que o algoritmo usado no *crate* é livre de corridas de dados, caso a regra de unicidade entre produtor e consumidor seja mantida. Mais uma vez, o sistema de *ownership* da linguagem Rust vêm para tornar idiomática a restrição imposta. A função `split` utiliza a fila inicial e cria os *endpoints* de produção e consumo, garantindo que ambos não serão duplicados ou apelidados repetidamente (*aliasing*). Através de

sanitizadores de *thread* (*ThreadSanitizer*), os autores da biblioteca provam em (RIVERA; LINDNER, M.; LINDGREN, 2018) que a codificação condiz com o esperado.

Rust se apoia nos fundamentos de concorrência da linguagem C/C++, descritos em (BATTY, 2014). Em modelo de memória relaxado, a menos que explicitamente indicado, o compilador não faz garantias sobre o ordenamento das instruções de leitura e escrita. Operações atômicas, informam abertamente que precisam de outras garantias, emitindo barreira (*fences*) capazes de desabilitar otimizações de compilação e hardware (RUST LANGUAGE, 2019m). Os valores de '`head`' e '`tail`' são protegidos por escritas e leituras *Acquire-Release*, como pode ser observado em (APARICIO, 2019i).

5.5 Considerações Finais

Esse capítulo apresentou o conteúdo relacionado às técnicas de programação concorrente para microcontroladores em Rust. Começou mostrando como a linguagem, por si só, não é capaz de impedir a existência de vulnerabilidades advindas do mal uso de recursos compartilhados, justificando a utilização da *RTFM-lang*. Terminou elucidando a importância de estruturas de dados e algoritmos adaptados a trabalhar em aplicações de tempo real. Cabe ressaltar que os detalhes relativos ao funcionamento interno da *framework* estão fora do escopo desse trabalho. Os pontos abordados ao longo do texto vieram para suprir a falta de documentação oficial apropriada.

Capítulo 6

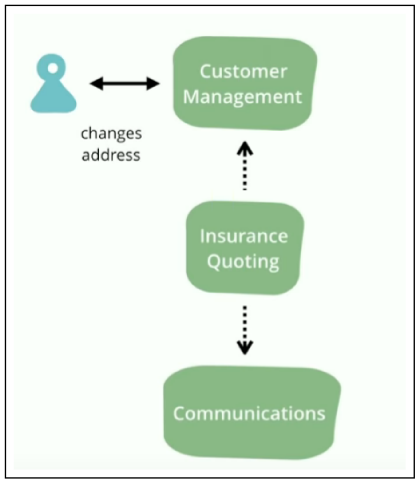
Extensões ao RTFM

Em (FAIRBANKS, 2010), os autores defendem que o sucesso para combater a escala e a complexidade dos sistemas, nas próximas décadas, está intimamente ligado à habilidade dos programadores em melhorarem seus arsenais. Não se deve esperar uma "bala de prata" capaz de eliminar todas as dificuldade enfrentadas no desenvolvimento de softwares, conforme elucidado por (BROOKS, 1986). É preciso incorporar ferramentas que ajudem a particionar, entender e abstrair corretamente a essência do problema. Os padrões arquiteturais podem ser uma dessas armas.

Enquanto alguns acreditam que a arquitetura deva ser o pivô estrutural do processo de desenvolvimento, outros defendem que ela surgirá naturalmente a partir de um bom design. O primeiro grupo prega a construção focada na arquitetura. O segundo tende a minimizar o elaboração arquitetural como uma atividade separada do desenvolvimento (FAIRBANKS, 2010). Nenhuma das vertentes se encaixa em todos os casos, a arquitetura de software deve ser usada quando ela for capaz de reduzir a complexidade.

A abordagem de baixo nível do RTFM, próximo ao hardware, traz eficiência à solução. Entretando, o design de baixo para cima nem sempre é recomendado para conter a complexidade. Como contribuição, o presente trabalho, propõe dois padrões de arquitetura, complementares ao modelo de eventos e recursos pregado pelo RTFM. O primeiro é baseado em filas de eventos e o segundo em máquinas de estado.

Figura 6.1: Sistema de uma seguradora utilizando um modelo de arquitetura convencional.



Fonte: (FOWLER, 2017)

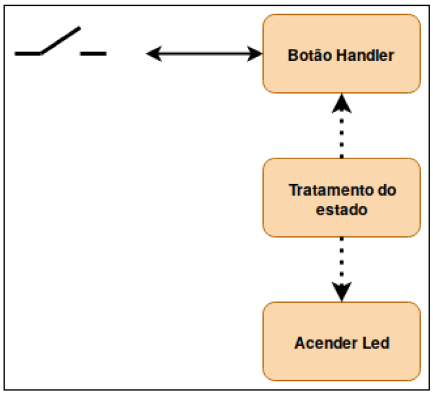
6.1 Filas de eventos

Imagine uma companhia seguros com um sistema semelhante ao mostrado na Figura 6.1. Ao solicitar uma mudança de endereço, o cliente provoca uma reação em cadeia na aplicação. O subsistema de gerenciamento de clientes, após alterar seus registros internos, precisa informar ao módulo responsável pelas cotações a mudança de endereço. Finda a análise sobre as consequência da alteração requerida, o subsistema de cotações deve tomar as ações cabíveis, como acionar o sistema de comunicação. Esse por sua vez, avisará o cliente sobre, por exemplo, uma mudança do valor cobrado.

Nota-se um alto grau de interligação entre os subsistemas. A parte mais genérica, o gerenciamento de dados pessoais, precisa conhecer detalhes de funcionamento dos subsistemas adjacentes. A regra de negócio está espalhada entre todos os componentes, formando uma trama difícil de ser mantida. Uma pequena alteração pode impactar em todos os componentes do sistema.

As arquiteturas norteadas por eventos propõem uma inversão de controle em relação ao padrão anterior. Ao invés de ressaltar a natureza procedural e sequencial entre os componentes (visto na Figura 6.1) busca evidenciar a reatividade das atividades. O estado da aplicação é reflexo dos processos pelos quais ela passou. Segundo (FOWLER, 2017), os quatro padrões arquiteturais orientados por eventos, mais usuais, em sistemas de informação são: *Event Queue*, *Event Carried State Transfer*, *Event Sourcing* e *Command Query Responsibility Segregation* (CQRS).

Figura 6.2: Fluxo de tratamento de ações seguindo uma arquitetura tradicional.



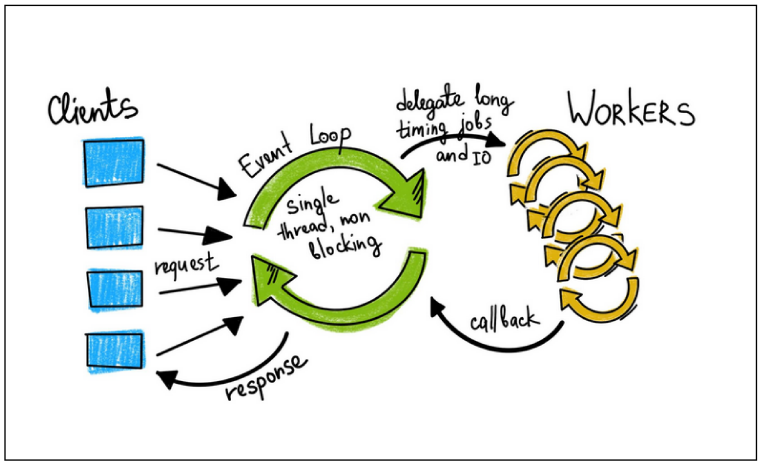
Fonte: Autoria própria

Em uma aplicação RTFM tradicional, o fluxo de controle está diluído, entre tarefas de hardware que reagem a eventos externos e tarefas de software criadas para lidar com as consequências do estímulo recebido. A sequência descrita acima se assemelha à estratégia adotada na construção do sistema de seguros (Figura 6.1). Suponha, por exemplo, que ao clique de um botão o manipulador de interrupção queira disparar uma série de *soft-tasks*, Figura 6.2. Para tal, ele precisa conhecer o fluxo futuro, como os protótipos das funções chamadas. É explícito na função produtora o escalonamento dos próximos passos.

Dentre os padrões arquiteturais difundidos, ao presente trabalho interessa a arquitetura orientada por eventos de notificação (*Event Queue*). Nela os componentes do sistema são desacoplados por uma fila de comandos. O evento funcionará como um objeto manipulável que encapsula informações e direciona os próximos passos. Incorporar filas de notificação é uma extensão natural do modelo subjacente. Agora, além de reagir aos estímulos (eventos) externos, via periféricos, as tarefas irão responder à eventos produzidos por outras tarefas. Ao contrário de (QUANTUM LEAPS, 2019) e (LEE, Edward A, 2003, 2002), a arquitetura proposta não prevê a criação de um sistema de atores, ainda é permitido compartilhar recursos entre tarefas, contanto que esses estejam protegidos pelo SRP.

Até então os termos “evento” e “comando” foram usados indistintamente, porém aqui cabe a distinção conceitual. Nos comandos, a função de origem é incisiva em relação ao comportamento do sistema, esperando ações concretas. Enquanto os eventos têm natureza mais informativa, cabe ao receptor a tomada de decisão.

Figura 6.3: Ilustração do padrão de design *Reactor*.



Fonte: (KIRILL, 2019)

Padrões de arquitetura clarificam apenas a estruturação de alto nível do sistema a ser construído. A interligação dos componentes e suas respectivas implementações são tratadas a nível de design e idioma. *Reactor* (SCHMIDT, 1994) e *Proactor* são os padrões de construção mais difundidos para a demultiplexação de filas de eventos. Em comum, possuem possuem a habilidade de multiplexar elementos mas se diferem na forma de despachá-los. No *Reactor* (Figura 6.3) os eventos são liberados sincronamente, ou seja, executam até a finalização, antes que o próximo possa começar. Já no *Proactor*, eles são disparados de forma assíncrona. Quando finalizadas as tarefas retornam o estado a um "manipulador de conclusão" *(completion-handler)*.

A solução proposta por esse texto apoia-se em eventos síncronos, portanto, se assemelha a um *Reactor*. Diferencia-se do original (Figura 6.3) por não possuir retorno de respostas aos clientes.

O *Reactor* é uma evolução do padrão "canais de eventos" (BUSCHMANN et al., 1996). Nos canais não há a ideia de demultiplexação, apenas trocas de informações entre produtores e consumidores. Diversos conceitos apresentados por esse padrão, como desacoplamento e múltiplos produtores, foram incorporados pelo *Reactor*.

O padrão *Reactor* pode ser visto como uma variação do padrão *Observer* (GAMMA et al., 1997). No *Observer*, os consumidores são atualizados automaticamente quando um tópico é alterado. No *Reactor*, os manipuladores são informados automaticamente quando ocorrem eventos. Em geral, o *Reactor* é usado para demultiplexar várias fontes de entrada para seus manipuladores de eventos associados, enquanto o *Observer* é, geralmente, associado apenas a uma única fonte de eventos.

Como quase todas as escolhas técnicas, os padrões de arquitetura norteados por eventos possuem desvantagens associadas. Como há uma inversão no fluxo de controle, é mais difícil depurar o código *event-driven*.

6.1.1 Implementação

O *Reactor* pode ser codificado de diversas formas, variando com o paradigma e a linguagem escolhida. Todavia, alguns tópicos trazem dificuldades comuns a todas as implementações. Segundo (SCHMIDT, 1994), são eles: sincronização, demultiplexação, semântica de entradas e saídas, flexibilidade e extensividade dos manipuladores de evento. Uma boa interpretação do *Reactor* deve abordá-los corretamente. Nas linguagens orientadas a objeto, para atingir o grau de flexibilidade necessário, recomenda-se o uso de funcionalidades como *templates*, herança e ligação dinâmica (SCHMIDT, 1994).

Como mostrado na Subseção 3.8.2, Rust possui uma interpretação particular em relação à orientação a objeto clássica. Idiomaticamente, nas aplicações embarcadas, as dificuldade acima serão tratadas com o uso de *trait objects*, primitivos do RTFM e as estruturas `heapless`.

Um evento, ou comando, será qualquer enumeração que se comporte com tal, ou melhor, que implemente o *trait* `isEvent`. Essa delimitação de comportamento resolve um dos problema, a multiplexação (Listagem 6.1). Usando esse traço como *trait object*, a fila segregadora pode aceitar qualquer tipo de elemento, contanto, que ele "seja um evento".

Listagem 6.1: Eventos será quaisquer objeto que implemente o 'trait marker isEvent'.

```
1  // ‘Trait marker‘ criado para informar informar ao compilador
2  // que determinado ‘enum‘ eh um ‘evento‘.
3  pub trait IsEvent { }
4
5  type EventObject = &’static IsEvent;
6
7  pub struct Event {
8      e: EventObject,
9  }
10 // Informando ao compilador que ‘Event‘ pode ser transferido
11 // entre ‘threads‘.
12 unsafe impl Send for Event {}
```

A demultiplexação dos eventos virá com a evolução do traço definido no Código 6.1. Na Listagem 6.2, ele será mais do que marcador, irá definir o procedimento a ser executado quando o evento for desenfileirado, ou seja, será responsável pela demultiplexação. Nessa nova versão, in-

terno ao `isEvent`, há o método `run`. Note que como não há uma implementação padrão para ele, qualquer objeto que deseje "ser um evento" precisará elaborar seu algoritmo de despacho. O objetivo da função `run` é escalonar *soft tasks*, as quais, realizarão as operações previstas no design da aplicação. Os detalhes internos variam com a lógica de negócio a ser implementada e, aqui, não possuem nenhum compromisso com a realidade.

`Run` recebe uma referência imutável do "Evento" e a estrutura de `spawn` da tarefa consumidora. Nesse modelo não há distinção sintática entre eventos e comandos, o desenvolvedor tem total liberdade de usá-los em conjunto, garantindo a legibilidade da solução. A Listagem 6.2 mostra a versão final do objeto de traço "Evento" e como adicionar comportamento '`isEvent`' os `enums` `LedAction` (linhas 23-40) e `ADC` (linhas 47-52).

Listagem 6.2: Principais conceitos empregados para construção do Reactor para aplicações RTFM.

```
1  pub trait IsEvent {
2      // Nessa funcao o ‘evento‘ deve definir o
3      // comportamento da demultiplexacao.
4      fn run(&self, spawn: idle::Spawn);
5  }
6
7  /// Acoes de comando sobre Leds.
8  #[derive(Debug)]
9  pub enum LedAction {
10     On([Led; 2]),
11     Off(Led),
12     ChangeDuty,
13 }
14
15 /// Informando ao compilador que ‘LedAction‘ eh um ‘evento‘ e
16 /// quais procedimentos devem ser tomados com o despacho do mesmo.
17 impl IsEvent for LedAction {
18     fn run(&self, spawn: idle::Spawn) {
19         // O match garante que as variacoes serao exultivamente testadas,
20         // o desenvolvedor deve explicitar os proximos passos para todas as
21         // variantes, mesmo que seja, as ignorar.
22         match self {
23             LedAction::On(leds) => {
24                 for led in leds {
25                     spawn.led_enable(led.clone()).unwrap();
26                 }
27             },
28             LedAction::Off(led) => {
29                 spawn.led_disable(led.clone()).unwrap();
30             },
31             LedAction::ChangeDuty => spawn.led_duty().unwrap(),
32         }
33     }
34 }
35
36 #[derive(Debug)]
37 pub enum ADC {
38     Read,
39 }
40
41 impl IsEvent for ADC {
42     fn run(&self, spawn: idle::Spawn) {
```

```
43          // para qualquer variante, a tarefa ‘adc_read‘ sera escalonada.
44          spawn.read_adc(true).unwrap();
45      }
46  }
```

A variável ’q’, na Listagem 6.3, é uma `heapless::Queue` de `Events`, o *trait object* para `isEvent`. Para adicionar um novo evento à fila, basta enfileirá-lo (`enqueue`) como ilustrado na linha 5. Eventos são consumidos desinfileirando (`dequeue`) itens da fila (linha 12).

Listagem 6.3: A variável ’q’ é uma fila de eventos.

```
1  // Fila de eventos com capacidade para 3 elementos.
2  let q: Queue<Event, U3> = Queue::new();
3
4  // enfileirando o evento ’led off’.
5  q.enqueue(
6            Event {
7          e: &LedAction::Off([Led::Yellow, Led::Red]),
8      }
9  );
10
11 // desinfileirando um elemento.
12 if let Some(item) = q.dequeue() {
13      item.e.run(spawn);
14 }
```

No padrão proposto, o disparo das tarefas de software está “escondido” atrás de um artifício de arquitetura, contudo, as regras de escalonamento são mantidas. O RTFM continuará analisando as prioridade e recursos compartilhados para escolher a próxima tarefa a entrar em execução. Além disso, continua garantindo que somente a quantidade de mensagens previamente definidas, `capacity`, estará disponível.

A inversão do fluxo de controle e separação de conceito facilita futuras manutenções e trocas. Adicionar um novo evento, um novo produtor, ou mesmo um novo ponto de consumo à uma atividade implica em trocas pontuais, afetando pouco no restante do sistema. Com uso extensivo de tarefas de software considera-se, desde o início, a operação com requisitos flexíveis de tempo.

6.1.2 Aplicação RTFM

A Listagem 6.4 apresenta uma aplicação RTFM usando a definição da *event queue* mostrada na subseção anterior. Aqui a fila de eventos, `Q`, é um recurso RTFM com a capacidade para 8 elementos. Aproveitando-se do modo `spsc`, ela será separada no produtor `P` e no consumidor `C` que também serão recursos da plataforma. `P` deve ser requerido por todas as tarefas que produzem

eventos. O protocolo SRP garantirá que o acesso compartilhado é seguro, isto é, sem *data-race*. Já `C` será um recurso exclusivo da função `idle`, acessado diretamente via `resources`.

Listagem 6.4: 'Reactor' embarcado em uma aplicação RTFM.

```
#[derive(Debug, PartialEq, Clone)]
pub enum Led {
    Yellow,
    Red,
    Green,
}

// Codigo omitido.
// Demultiplexacao dos eventos segue o codigo apresentado na secao anterior.
pub trait IsEvent ... ;

#[app(device = narc_hal::stm32l052)]
const APP: () = {
    // Para contorna a necessidade de instanciacao previa e constante
    // de todas as variaveis ‘static‘, a ‘queue‘ foi escondida atras
    // de um ‘Option‘.
    static mut Q: Option<Queue<Event, U8>> = None;

    static mut P: Producer<’static, Event, U8> = ();
    static mut C: Consumer<’static, Event, U8> = ();

    static mut LED_RED: PA7<Output<PushPull>> = ();
    // Demais perifericos.
    ...;

    static mut WT: TimerWheel<bool, U10, U1> = ();

    #[init(resources = [Q])]
    fn init() {
        // Inicializacao dos perifericos.
        ...;

        *resources.Q = Some(Queue::new());
        let (p, c) = resources.Q.as_mut().unwrap().split();
        P = p;
        C = c;
    }

    #[idle(resources = [C], spawn = [led_enable, ..., adc_read])]
    fn idle() -> ! {
        loop {
            ...;

            // Execucao de um ‘item‘ desenfileirado.
            if let Some(item) = resources.C.dequeue() {
                item.e.run(spawn);
            }
        }
    }

    #[interrupt(resources = [P, ...])]
    fn TIM6_DAC() {
        // tratamento anti flutuacao do botao.
        let b1: u16 = if resources.B1.is_high() { 1 } else { 0 };
        *resources.B1_COUNTER = (*resources.B1_COUNTER << 1) | b1 | 0xe000;

        if *resources.B1_COUNTER == 0xf000 {
            // Enfileira o comando de acender os Led ‘Yellow‘ e ‘Red‘.
            resources.P.enqueue(
                Event {
```

```
61                      e: &LedAction::On([Led::Yellow, Led::Red]),
62                  }
63              );
64          }
65      }
66
67      #[exception(resources = [P, WT])]
68      fn SysTick() {
69          // Usando a timer-wheel para adiar a o enfileiramento do
70          // ‘LedAction::Off‘ em 3 segundos.
71          let mut has_func = false;
72          {
73              let to_run = resources.WT.tick();
74              while let Some(_) = to_run.pop() {
75                  has_func = true;
76              }
77          }
78
79          if has_func {
80              // DONE enqueue disable led
81              resources.P.enqueue(
82                  Event {
83                      e: &LedAction::Off([Led::Yellow]),
84                  }
85              );
86          }
87      }
88
89      /// Capacity > 1
90      #[task(resources = [WT, LED_YELLOW, LED_RED], capacity = 2)]
91      fn led_enable(led: Led) {
92          if led == Led::Yellow {
93              resources.LED_YELLOW.set_high();
94          }
95          if led == Led::Red {
96              resources.LED_RED.set_high();
97          }
98
99          // usando a timer-wheel para adiar o enfileiramento da aÃ§Ã£o
100         // ‘LedAction::Off‘.
101         let _ = resources.WT.schedule(3, true);
102     }
103
104     #[task(resources = [LED_YELLOW, LED_RED])]
105     fn led_disable(led: Led) {
106         // Codigo omitido. Semelhante a tarefa ‘led_enable‘.
107     }
108
109     // As demais ‘sotf-tasks‘ serao omitidas, para facilitar a compreensao
110     // dos conceitos mostrados acima.
111     ...;
112 }
```

Na Listagem 6.4 é possível analisar a estrutura completa de um programa *RTFM-Reactor*. Ao clique do botão (linha 57), o comando `LedAction::On([Led::Yellow,Led::Red])` é enfileirado (linha 59). Três segundos após o '`led_enable(led: Led)`' (linha 91-102), a indicação '`LedAction::Off(Led:: Yellow)`' (linha 81) é produzida. Partes do código que não iriam facilitar a compreensão do leitor foram omitidas. A implementação completa e outros exemplos de utilização estão disponíveis em (SILVA, 2019c). O processo de demultiplexação usado pela aplica-

ção foi apresentado na Listagem 6.2, por isso, também foi omitido.

A Listagem 6.4 não representa nenhum problema real. Foi criado apenas para explorar a construção de uma aplicação norteada por eventos de baixo e alto nível. Ele demonstra as diversas interações possíveis e detalha o consumo e criação de eventos, além de elucidar o uso de uma *timer-wheel* como mecanismos de adiamento de atividades.

6.2 Máquinas de estado

Ainda na tendência de arquitetura orientada a eventos, um padrão muito aplicado em sistemas embarcados é a construção através de máquina de estados (*State Machine* - SM). Esse paradigma faz sentido quando o comportamento da aplicação pode ser separado em estados distinguíveis. A concepção a partir de SMs ajuda a reduzir a complexidade acidental, criando um nível de abstração fundamentado em estados, transições e ações.

As máquinas de estados são modelos de computação para sistemas discretos. Nelas, cada reação mapeia valores de entrada em valores saídas, dependendo do estado atual. Elas são particularmente naturais aos sistemas físicos computacionais se encaixam bem em seu comportamento reativo. As interações em uma SM não precisam resultar em saídas físicas, entretanto, aos CPS interessa, principalmente, modelos capazes de afetar o ambiente no qual estão imersos, (LEE, Edward Ashford; SESHIA, 2012). Em especial, quando a quantidade de estados possíveis a um modelo é finita, ele define uma máquina de estados finita (*Finite State Machine* - FSM).

Não há formalmente nenhuma ligação temporal às máquinas de estado. Elas reagem a eventos, não se importando com a origem dos mesmos. Por exemplo, é irrelevante à sua definição formal, o tempo de duração de um determinado estado (LEE, Edward Ashford; SESHIA, 2012). Características como essa podem ser incorporados aos design da solução, todavia, não são atributos formais de uma máquina de estados.

Para entender o funcionamento desse modelo de computação é necessário compreender seus principais conceitos e implicações. O estado (*state*) de um sistema é sua condição em um determinado ponto do tempo. Formalmente, o estado codifica tudo sobre o passado da aplicação que pode afetar seu comportamento no presente ou no futuro. Em geral, ele interfere em como o sistema reage às entradas. Essa é, aliás, a propriedade cerne da modelagem por SMs.

A dinâmica da aplicação é governada pela transição (*transition*) entre os estados, sendo essa dirigida por funções de atualização (representação matemática) ou "predicados guarda" (máquina descrita graficamente). Quando, em particular, os estados inicial e final são coincidentes a transição é conhecida como *self-transition*.

O último pilar de abstração é a ação, nome convencionalmente dado ao processo de transformação das entradas, produzindo saídas. Por definição, as máquinas de estado são sistemas causais, ou seja, a saída do passo atual depende de dados passados e da entrada atual. Quando não há uma definição explícita do mapeamento entrada-saída é dito que a ação está ausente. O momento em que a "ação" é efetuada varia com o modelo de SM empregado. Historicamente, os dois principais formatos são *Mealy* e *Moore*.

Nas máquinas *Mealy* (MEALY, 1955) a saída é dependente da entrada e do estado corrente, por isso, realizam ações quando uma transição é efetuada. Nelas nenhuma saída é produzida a menos que uma transformação de estado esteja explicitamente envolvida.

Nas máquinas *Moore* (CHURCH, 1958), a saída é uma propriedade do estado corrente e não de uma transição. As entradas determinam a dinâmica da aplicação, porém não afetam imediatamente as saídas do sistema. A independência à entrada corrente, subclassifica as máquinas de estado *Moore* em estritamente causal.

Vale ressaltar que ambos os modelos representam comportamentos discretos, logo, suas operações são baseadas em uma sequência discreta de reações. Elas são traduzíveis entre si, qualquer máquina *Moore* pode ser traduzida para uma *Mealy* equivalente. Qualquer *Mealy* pode se transformar em uma *Moore* quase análoga, com a diferença que a saída será produzida após a reação ao invés de durante a transição.

As máquinas *Mealy,* normalmente, resultam em soluções mais compactas, menos estados são necessário para representar o mesmo comportamento. Além disso, conveniências técnicas (facilidade de codificação e ajuste ao RTFM) levaram o presente trabalho a optar por essa vertente, o restante do texto se refere a essa família de modelos.

6.2.1 Representação Matemática

Matematicamente uma FSM pode ser representada por uma tupla com 5 elementos, Equação 6.1 (LEE, Edward Ashford; SESHIA, 2012). O campo 'Estado' delimita o conjunto dos possíveis estados

da aplicação. As 'Entradas' ($x(n)$) agrupam as variações de entrada. Semelhantemente, as 'Saídas' ($y(n)$) colecionam os possíveis valores de saída. A função de atualização define formalmente o comportamento das transições, mapeia 'Entradas' e 'Estado' para 'Saídas' e próximos 'Estado', segundo a Equação 6.2. Os 'EstadosIniciais', intuitivamente, apresentam as possíveis condições de *reset* da aplicação, $s(0)$.

Os valores específicos de cada posição variam com o problema a ser solucionado. Como a esse trabalho interessa a representação gráfica das SMs, mais detalhes serão omitidos.

$$(Estados, Entradas, Saídas, Atualização, EstadoInicial) \tag{6.1}$$

$$(s(n+1), y(n)) = atualização(s(n), x(n)) \tag{6.2}$$

6.2.2 Representação Gráfica

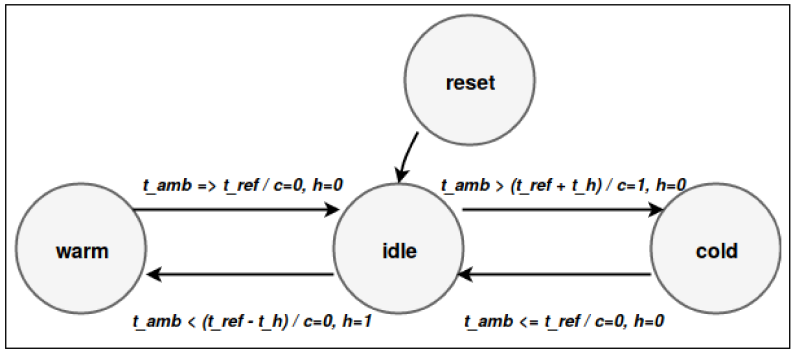
Em sua representação gráfica, uma máquina de estados é um diagrama de elipses e arcos. A título de ilustração, a Figura 6.4, apresenta um sistema de ar-condicionado projetado segundo uma SM. Os estados *Warm*, *Idle* e *Cold*, são exibidos como elipses nomeadas. Já as transições entre eles são arcos ligando a elipse inicial à elipse final. Ao redor das transições estão os predicados guarda e as ações, `guarda / ações`, como em: `t_amb => t_red / c=0, h=0`.

A aplicação transita entre os três estados disponíveis de acordo com a diferença entre a temperatura ambiente (`t_amb`) e temperatura de referência definida pelo usuário (`t_ref`). A entrada `t_h`, temperatura de histerese, evita que ocorram trepidações, ou trocas abruptas de estado. O sistema está em `Idle`, equilíbrio, quando a diferença entre a temperatura desejada e a obtida é menor do que a histerese do sistema. Comportamento esse, evidenciado nas transições `Idle -> Cold` e `Idle -> Warm`.

Ao transitar para um novo estado, uma "ação" é tomada. Nesse exemplo, o valor das variáveis `c` (*cold*) e `h` (*heat*) são atribuídos conforme a necessidade do sistema em esfriar ou esquentar o ambiente. Nível lógico alto (`1`) indica atividade, nível lógico baixo (`0`) ausência de atividade. Em uma aplicação real essas variáveis nortearão os sistemas de arrefecimento e aquecimento.

O sistema fica no estado `Cold`, resfriando o ambiente, até que a temperatura atinja o nível

Figura 6.4: Representação gráfica de uma máquina de estados *Mealy* para um sistema de ar-condicionado.



Fonte: Autoria própria

esperado, isto é, enquanto o predicado `t_amb <= t_ref` não retorna verdadeiro. Similarmente, a aplicação fica em modo de aquecimento enquanto a temperatura ambiente não condiz com o espectativa do usuário, ou melhor, até que `t_amb => t_ref`.

A escolha por um exemplo tão simples e comum foi proposital. Com isso, espera-se que o leitor possa focar nas soluções propostas adiante sem se preocupar em entender o problema.

6.2.3 Determinismo e receptividade

Outras duas características secundárias são cruciais para a classificação das máquinas de estado. São elas: o determinismo e a receptividade. Uma SM é receptiva, caso nessa, para cada um de seus estados existe pelo menos uma transição possível. Em outras palavras, a receptividade garante que a máquina sempre está pronta para reagir a uma entrada, sem ficar presa em algum estado indefinido. Lembrando que, na notação gráfica, todos os estados possuem uma auto-transição implícita: caso nenhuma transformação seja habilitada para tal entrada, o sistema permanece em seu estado atual.

Uma SM é dita determinística quando, para cada um dos seus estado, há no máximo uma transição habilitada para cada valor de entrada. A definição formal de uma FSM garante que ela é determinística já que a atualização é controlada por uma função, não por um mapeamento de um para muitos.

A notação gráfica, no entanto, não evidencia essa restrição, transferindo para cada estado a análise do determinismo. Esse será determinístico caso suas funções guardas não se sobreponham.

Graficamente, uma transição não determinística é colorida em vermelho. Máquinas com mais de um estado inicial não são determinísticas.

Uma máquina determinística tem seu comportamento completamente definido a partir da sequência de entrada, as não determinísticas não possuem a mesma propriedade. Entretanto, cabem em modelagem mais complexas, abstraindo característica irrelevantes do ambiente, (LEE, Edward Ashford; SESHIA, 2012).

6.2.4 Implementação

No que se refere à implementação de um sistema de máquinas de estado, diversas soluções de design são possíveis. Enquanto algumas construções evidenciam os requisitos formais de uma FSM, em outras, manter-se dentro do esperado é um dever do programador.

Soluções mais robustas partem para a primeira alternativa. Nelas, as abstrações criadas certificam que as propriedades de uma SM são verdadeiramente atendidas, criando um nível de segurança. Um código apoiado em tais conceitos não irá incorrer em equívocos ao modelo formal.

Por exemplo, em uma SM, há, sempre, apenas um estado vigente; um bom sistema de construção promete que esse requisito é de fato verdadeiro. Da mesma forma, como as transições entre os estados são previamente definidas, uma implementação robusta, controla a semântica das transições assegurando que somente operações válidas serão tomadas durante a execução.

A codificação mostrada na Listagem 6.5 procura atender os pontos acima além de certificar que a máquina sempre inicializa em um dos estados iniciais informados. Cabe ressaltar que, onde possível, soluções estáticas de verificação foram empregadas. Com isso, o compilador está encarregado de analisar o sistema implementado em termos sintáticos e semânticos.

Falhas em atender as invariâncias das máquinas de estado são convertidas em erros ligados ao sistema de tipos, propriedade e empréstimo. Estados, transições e ações serão estruturas com características e comportamento próprios. A implementação está repleta de concepções já apresentadas ao longo desse texto, como genéricos monomorfizados e enumerações. Nessa primeira seção, através de uma codificação completamente manual, os principais conceitos de design são apresentados. Os próximos tópicos abordam como essa solução foi transformada em uma linguagem de domínio específico.

Uma “máquina de estado” é uma estrutura que implementa o *trait* `Machine`. Esse traço de-

fine conceitualmente o que é uma '*State Machine*', especificando seus tipos associados e métodos. `State` e `Event` são os valores relacionados à uma SM. Eles, respectivamente, indicam o estado atual e o evento que resultou nele. Os métodos `state` (linha 17) e `trigger` (linha 18), permitem o acesso a essas informações internas (Listagem 6.5).

Listagem 6.5: Definição do comportamento básico de uma ‘Machine‘.

```
1  /// ‘Trait Marker‘, indica que tal ‘struct‘ eh um ‘State‘.
2  pub trait State {}
3
4  pub trait InitialState {}
5  pub trait Event {}
6
7
8  /// Definicao do comportamento de maquina de estados.
9  /// Agrupa: estado atual e a transicao que levou a tal.
10 pub trait Machine {
11     type State: State;
12     type Event: Event;
13
14     fn state(&self) -> Self::State;
15     fn trigger(&self) -> Option<Self::Event>;
16 }
17
18 pub trait Initializer<S: InitialState> {
19     /// Representa a ‘Machine‘ que o inicializador implementado deve retornar.
20     /// Como nao ha estado anterior, a ultima transicao eh a
21     /// ausencia de um evento, ‘NoneEvent‘.
22     type Machine: Machine<State = S, Event = NoneEvent>;
23     fn new(state: S) -> Self::Machine;
24 }
25
26 /// Define como implementar uma transicao entre dois eventos.
27 pub trait Transition<E: Event> {
28     type Machine: Machine;
29
30     /// ‘transition‘ consome o estado atual e retorna um nova
31     /// ‘machine‘, baseado no evento realizado.
32     fn transition(self, event: E) -> Self::Machine;
33 }
34
35 /// Converte a instancia de uma ‘Machine‘ em um enum.
36 /// ‘Enums‘ podem ser quebrados em padroes.
37 pub trait AsEnum {
38     type Enum;
39
40     /// ‘as_enum‘ consume a instancia atual da maquina de estados e retorna a
41     /// variante do ‘enum‘ que representa o estado atual.
42     fn as_enum(self) -> Self::Enum;
43 }
44
45 /// Unico ‘evento‘ padrao, o restante depende da solucao construida.
46 pub struct NoneEvent;
47 impl Event for NoneEvent {}
```

Um 'estado' é qualquer estrutura que implemente o traço marcador `State` (linha 2). Transições são controladas por 'eventos' e essas são quaisquer `struct` que implemente o *trait* `Event` (linha 5). A função '`transition`' (linha 32), entre dois estados, só existe quando há um evento válidos entre

os mesmos. Tentar converter `Warm` para `Cold` através de `toCool` resulta em erro de compilação (linha 10, Listagem 6.7). A transformação consome a instância atual (`transition(self, ...)`). Após a transição o estado anterior não existe mais, duplo uso, termina em erro (linha 15, Listagem 6.7).

Convenientemente, ao contrário da notação gráfica, aqui os eventos também são nomeados. A transição entre `Idle` e `Warm` se chama `ToHeat`, já `Idle -> Cold` é conhecida como `ToCool`. A volta para o estado de equilíbrio indica cessão da atividade de aquecimento (`StopHeat`) ou resfriamento (`StopCool`). Note na Listagem 6.6 (linhas 16, 20, 24, 18), que todas as quatro estruturas implementam o traço de indicação de evento. Apenas o 'evento ausente' (`NoneEvent`) é implementado por padrão para todas as SMs (linha 49, Listagem 6.5).

Como indicado em seu diagrama (Figura 6.4) esse sistema possui apenas uma opção de estado inicial, começando em '`Idle`'. Essa `struct`, por conseguinte, é a única a implementar o traço `InitialState` (linha 12, Listagem 6.6). Por padrão, todas as SM se inicializam a partir do evento especial `NoneEvent`. O construtor `new` interno ao *trait* genérico `Initializer` é monomorfizado para todas as estruturas que implementam `InitialState`, como mostrado na Listagem 6.6.

Listagem 6.6: Implementação manual de um sistema de máquina de estado para controlar a temperatura ambiente.

```
1  // Estado validos para a maquina de estado que representa o
2  // sistema de ar-condicionado.
3  pub struct Idle;
4  impl State for Idle;
5  pub struct Cold;
6  impl State for Cold;
7  pub struct Warm;
8  impl State for Warm;
9
10 // Implementado o traco de marcacao ‘InitialState‘ para o estado ‘Idle‘,
11 // indicado ser um estado do ‘reset‘ do sistema.
12 impl InitialState for Idle {}
13
14 // Evento que representa a transicao Idle -> Warm.
15 pub struct ToHeat;
16 impl Event for ToHeat {}
17
18 // Idle -> Cold
19 pub struct ToCool;
20 impl Event for ToCool {}
21
22 // Cold -> Idle
23 pub struct StopCool;
24 impl Event for StopCool {}
25
26 // Warm -> Idle
27 pub struct StopHear;
28 impl Event for StopHeat {}
29
```

```rust
// ‘Enum‘ equivalente.
pub enum Variant {
    InitialIdle(Machine<Idle, NoneEvent>),
    WarmByToHeat(Machine<Warm, ToHeat>),
    IdleByStopHeat(Machine<Idle, StopHeat>),
    ColdByToCool(Machine<Cold, ToCool>),
    IdleByStopCool(Machine<Idle, StopCold>),
}

// Implementado o inicializador generico ‘Initializer‘ a todos
// os estado que satisfacam a condicao de ser um ‘InitialState‘.
impl<S: InitialState> Initializer<S> for Machine<S, NoneEvent> {
    type Machine = Machine<S, NoneEvent>;
    fn new(state: S) -> Self::Machine {
        Machine(state, Option::None)
    }
}

// Implementa ‘AsEnum‘ para a maquina em estado ‘Idle‘ vindo do reset.
impl AsEnum for Machine<Idle, NoneEvent> {
    type Enum = Variant;
    /// Retorna a variante ‘InicialIdle‘
    fn as_enum(self) -> Self::Enum {
        Variant::InitialIdle(self)
    }
}
impl AsEnum for Machine<Warm, ToHeat> {
    type Enum = Variant;
    fn as_enum(self) -> Self::Enum {
        Variant::WarmByToHeat(self)
    }
}
impl AsEnum for Machine<Idle, StopHeat> {
    ...;
}
impl AsEnum for Machine<Cold, ToCool> {
    ...;
}
impl AsEnum for Machine<Idle, StopCold> {
    ...;
}

// Implementa a relacaio de transicao ‘ToHeat‘ a partir de ‘Idle‘
// (vindo de qualquer evento ‘E‘) para Warm.
impl<E: Event> Transition<ToHeat> for Machine<Idle, E> {
    type Machine = Machine<Warm, ToHeat>;
    // Consome a instancia anterior e retorna uma nova ‘Machine(Warm, ToHeat)‘.
    fn transition(self, event: ToHeat) -> Self::Machine {
        Machine(Warm, Some(event))
    }
}
impl<E: Event> Transition<StopHeat> for Machine<Warm, E> {
    type Machine = Machine<Idle, StopHeat>;
    fn transition(self, event: StopHeat) -> Self::Machine {
        Machine(Idle, Some(event))
    }
}
impl<E: Event> Transition<ToCool> for Machine<Idle, E> {
    ...;
}
impl<E: Event> Transition<StopCold> for Machine<Cold, E> {
    ...;
}
```

O *trait* `AsEnum` define a conversão de uma máquina para seu `enum` equivalente. Cada vari-

ante desse, está ligada a uma das possíveis nuances da SM, concentra o estado atual e a última transição. Como vantagem ele pode ser "quebrado" em padrões, através de `match` e `if let`. A Listagem 6.6 (linhas 31-37) apresenta o `enum Variant`, a representação "enumerada" do sistema de ar-condicionado. Por exemplo, caso a temperatura ambiente esteja abaixo do desejado, sistema em modo de aquecimento, ao converter sua '`sm`' para `enum` o resultado será a variação `WarmByToHeat`. Esse molde (`<State>By<Event>`) é válido para todas as outras transições exceto a inicial. Sintaticamente, a máquina em pós *reset* fica em `InitialIdle` (`Initial<State>`). Note que a transformação para `enum` consome a instância anterior, evitando *aliasing*.

Listagem 6.7: Aplicação consumindo a biblioteca de construção de máquina de estado definida acima.

```
1  let sm = Machine::new(Idle);
2  // error[E0277]: the trait bound ‘Sm::Warm: sm::InitialState‘ is not satisfied
3  let sm_erro = Machine::new(Warm);
4  //             ^^^^^^^^^^^^^ the trait ‘sm::InitialState‘ is not
5  //                                          implemented for ‘Sm::Warm‘
6
7  let sm_warm: Warm = sm.transition(ToHeat);
8
9  // error[E0308]: mismatched types
10 let sm_cold = sm_warm.transition(ToCold);
11 //                               ^^^^^^ expected struct ‘Sm::StopHeat‘,
12 //                                        found struct ‘Sm::ToCool‘
13
14 // error[E0382]: use of moved value: ‘sm‘
15 let sm_cold = sm.transition(ToCool);
16 //             ^^ value used here after move
17
18 match new_sm.as_enum() {
19
20 }
```

O design apresentado acima é uma padrão já difundido na comunidade Rust. Foi influenciado pela postagem (ANA HOBDEN, 2016) e está disponível em forma de DSL no *crate* `Sm` (MERTZ, 2019). Variações dele são muito usadas na construção de jogos e se apresenta com uma forma simples e idiomática de construção máquinas de estado em Rust.

Infelizmente, ele não se encaixa muito bem nas aplicações RTFM. Caso a '`SM`' seja um recurso compartilhado, cada uma das tarefas que o requer recebe apenas uma referência mutável (`&mut 'static`), não a propriedade do objeto. Por consequência, ele não pode ser consumido, como na função de transição tradicional. Ao mesmo tempo, simplesmente alterar o parâmetro de recepção do método `transition` para `&self` permitiria a ocorrência de *aliasing*. Como contribuição, o presente trabalho apresenta uma solução para construção de máquina de estados inspirada nos

conceitos acima, adaptada à executar em aplicação RTFM-Rust. Outro fator diferencial é a inclusão de funções guardas explícitas, aproximando-se da notação gráfica ideal. Em (MERTZ, 2019) não há nenhuma menção aos predicados de ativação, eles são considerados lógica de negócio, ou seja, o controle das transição está misturado à aplicação.

6.2.5 Linguagem SM

Simplicidade, segurança e rapidez, nortearam o desenvolvimento dessa linguagem de domínio específico. Apenas o controle do determinismo foi postergado ao *runtime*, todas as outras validações são estáticas e não implicam em perdas durante a execução. A sintaxe é simples, na qual, abertura e fechamento de chaves indicam o início e fim das seções. A Listagem 6.8 apresenta o mesmo sistema de ar-condicionado, usado ao longo desse texto, descrito através da DSL.

Listagem 6.8: Criação de uma máquina de estados através da DSL implementada.

```
sm!{
    Sm {
        GuardResources {
            { t_amb: U32 }
            { t_ref: U32 }
            { t_h: U32 }
        }
        ActionResources {
            { c: LedGreen }
            { h: LedRed }
        }
        InitialStates { Idle }
        ToHeat { Idle => Warm }
        StopHeat { Warm => Idle }
        ToCool { Idle => Cold }
        StopCold { Cold => Idle }
    }
}
```

A *macro* `sm` é a base dessa DSL. Ela é responsável por expandir o Código 6.8 em operações Rust convencionais. Os aspectos construtivos e as *macros* procedurais são apresentados na Seção 6.3. Para essa primeira seção, interessa sua percepção pela ótica usuário. O primeiro nível, nesse caso `Sm`, identifica cada máquina e modulariza seus componentes. Essa técnica permite a criação de várias SMs, não conflitante, conforme a Listagem 6.9.

Listagem 6.9: Definição de duas máquinas de estado simultaneamente.

```
sm!{
    Maquina1 { ... }
    OutraMaquina { ... }
}
```

Cada transição é representada por uma seção, conforme o modelo: `<Evento>{<EstadoInicial> => <EstadoFinal>}`, como `ToHeat{Idle => Warm}`. Eventos que não atendem a esse formato são negados. Não é possível, por exemplo, ligar um estado inicial a dois estados finais por uma única operação. As transições devem ser unicamente nomeadas, repetições terminam em erro de compilação.

Os possíveis estados iniciais são enumerados no segmento `InitialState`. Caso haja mais de uma opção, eles devem ser separadas por vírgula.

A seção 'GuardResources' delimita as entradas do sistema, seus nomes e respectivos tipos. O setor 'ActionResources' tem funcionalidade semelhante, informando sobre variáveis de ação. Tanto as variáveis de entrada quanto as de ação podem ser de qualquer tipo. Seja uma primitiva, como os inteiros não sinalizados usados até aqui, ou tipos mais complexos como estruturas de abstração de hardware. Lembrando que, caso a variável seja um recurso de plataforma, seu acesso pode requerer o uso de uma seção crítica.

As seções de recurso ponderadas acima definem a criação do *trait* `ValidEvent` durante a expansão da *macro* `sm`. Esse traço é específico para a SM e assegura que todas a transições disponíveis atendem à semântica de uma máquina *Mealy*, `guarda / ação` .

Para cada evento de transição definido, é necessário atestar sua validade, isto é, implementar-lhe o traço `ValidEvent`. O não cumprimento dessa premissa resulta em um erro de compilação. O trecho de código (Listagem 6.10) apresenta como esse requisito foi executado para o sistema de ar-condicionado.

Seguindo a definição formal, um predicado guarda deve sempre retornar um valor booleano. A DSL irá, em compilação, garantir que tal requisito é atendido. Para isso, o método `is_enabled` impreterivelmente recebe todas as entradas informadas e retorna um booleano.

As ações têm atuação no mundo físico, realizado pela a função `action`. Ela não deve retornar nenhum valor ao restante do sistema. Além disso, o compilador garante que todas as função de ação estão cientes das variáveis nas quais podem atuar.

Listagem 6.10: Definindo o comportamento dos eventos de transição. A função 'is_enabled' descreve o predicado guarda e 'action' expõe as ações de tal transição.

```
1 // Definindo o comportamento da transicao `ToHeat`.
2 impl Sm::ValidEvent for Sm::ToHeat {
3     // O predicado guarda sempre retorna um booleano,
4     // e opera sobre as variaveis informadas em `GuardResources`.
```

```
    fn is_enabled (t_amb: U32, t_ref: U32, t_h: U32) -> bool {
        t_amb < (t_ref - t_h)
    }
    // A acao recebe as variaveis de acao e opera sobre elas, nao
    // retorna nada ao sistema.
    fn action (c: LedGreen, h: LedRed) {
        c.set_high();
        h.set_low();
    }
}

impl Sm::ValidEvent for Sm::StopHeat {
    fn is_enabled (t_amb: U32, t_ref: U32, _t_h: U32) -> bool {
        t_amb >= t_ref
    }
    fn action (c: LedGreen, h: LedRed) {
        c.set_low();
        h.set_low();
    }
}

impl Sm::ValidEvent for Sm::ToCool {
    ...;
}

impl Sm::ValidEvent for Sm::StopCold {
   ...;
}
```

Ao contrário da seção anterior, aqui a aplicação não usará diretamente as transições de baixo nível (como `sm.transition(ToHeat)`). A nova sintaxe simplificada torna o funcionamento interno ainda mais abstrato ao usuário, enquanto mantém as mesmas garantias. O procedimento '`eval_machine`' deve executar a cada reação do sistema. Esse método, produzido durante a expansão por *macros*, adaptada-se à estrutura da máquina de estados pertinente. Ele recebe todos os '`Resources`', em ordem de definição, e efetua a '`transition`' habilitada para o estado corrente.

O método '`eval_machine`' retorna o `enum Result<(), ()>`. Quando apenas um dos eventos de transição do estado corrente é verdadeiro para a entrada atual, a transformação é efetuada internamente e, a variante `Ok(())` é retornada. Quando não há determinismo, a opção `Err(())` indica que as guardas estão sobrepostas e que nenhuma ação interna foi efetivada. A DSL ajuda o programador a entender o custo de uma operação não determinística, porém, não impõe a forma como essa deva ser tratada.

Caso seja um sistema crítico, pode-se negar as ambiguidades, com `eval_machine(...).unwrap()`. Assim, se houver alguma transição não determinística, o programa irá falhar, `panic!`. Um tratamento mais conservador também é possível. A aplicação pode, por exemplo, resetar para algum estado de segurança; ou de forma insegura, efetivar a transição manualmente. A Listagem 6.11 demonstra as duas vertentes, ambas, idiomáticas à linguagem Rust.

Listagem 6.11: Avaliação de uma máquina de estado. Contrastando suas opções de tratamento de erro.

```
// ‘panic!‘ caso haja nao-determinismo.
sm_old.eval_machine(t_amb, t_ref, 100, led_green, led_red).unwrap();

// Tratamento mais rebuscado de erro, reinicializacao da maquina em caso
// de transicao nao deterministica.
let result = sm_old.eval_machine(t_amb, t_ref, 100, led_green, led_red);
match result {
    Ok(_) => (),
    Err(_) => {
        *sm_old = Sm::Machine::new(Sm::Idle).as_enum();
    }
}
```

O Programa 6.12 apresenta uma solução RTFM usando as definições mostradas acima. Partes não fundamentais foram omitidas e as Listagem 6.8 e 6.10 completam a aplicação. As escolhas técnicas foram pensadas para evidenciar como a solução proposta é, no fim, uma extensão idiomática de uma aplicação RTFM.

Listagem 6.12: Aplicação RTFM usando máquinas de estado.

```
#![no_main]
#![no_std]

// Importacoes.
use rtfm::app;
...;
use sm::sm;

// Apelidando tipos proprios.
type LedGreen<'a> = &'a mut PA5<Output<PushPull>>;
type LedRed<'a> = &'a mut PA7<Output<PushPull>>;
type U32 = u32;
type BoolRTFM<'a> = &'a mut rtfm::Exclusive<'a, bool>;
type Bool = bool;

#[app(device = narc_hal::stm32l052)]
const APP: () = {
    static mut SM: Sm::Variant = ();
    static mut ADC: narc_hal::stm32l052::ADC = ();
    ...;
    static mut TIM6: timer::Timer<TIM6_p> = ();

    #[init]
    fn init() {
        // Configuracao dos perifericos.
        ...;

        use Sm::*;
        let sm = Machine::new(Idle).as_enum();

        // Configuracao timers.
        ...;

        SM = sm;
        // Demais recursos atrasados.
    }

```

```
38      #[idle()]
39      fn idle() -> ! {
40          loop {
41              wfi();
42          }
43      }
44
45      #[exception(resources = [SM, ADC, ADC_VALUE], priority = 4)]
46      fn SysTick () {
47          *resources.ADC_VALUE = resources.ADC.read();
48
49          cortex_m::peripheral::SCB::set_pendsv();
50      }
51
52      #[exception(resources = [SM, ADC_VALUE, LED_GREEN, LED_RED, T_REF],
53                  priority = 1)]
54      fn PendSV() {
55          use Sm::MachineEvaluation;
56
57          let led_green = resources.LED_GREEN;
58          let led_red = resources.LED_RED;
59
60          let t_amb = resources.ADC_VALUE.lock(|adc_value| *adc_value);
61          let t_ref = resources.T_REF.lock(|t_ref| *t_ref);
62
63          resources.SM.lock(|sm_old| {
64              sm_old.eval_machine(t_amb, t_ref, 100,
65                                  led_green, led_red).unwrap();
66          });
67      }
68
69      #[interrupt(resources = [TIM6, B1_COUNTER, B1, B2_COUNTER, B2, T_REF],
70                  priority = 2)]
71      fn TIM6_DAC() {
72          // Tratamento dos botoes B1 e B2 (‘debouncing‘).
73      }
74  };
```

Para evitar perdas de *deadline*, a avaliação temporal da máquina (a cada 1 milisegundo) não será “pendurada” no *tick* do sistema. O ’SisTick’, interrupção altamente prioritária, colocará PendSV em modo pendente. O PendSV é a mais *soft* das *hard-tasks*; quando configurado em baixa prioridade, é usado para postergar atividade menos críticas. Os botões B1 (+) e B2 (-) gerenciam a temperatura de referência, seus cliques e *debouncing* são controlados pela unidade 6 do *timer* (TIM6).

Uma versão mais completa dessa aplicação está disponível em (SILVA, 2019d). Nela “rodas de tempo” são usadas para emular a presença de contatores, nela, o sistema deve “ficar pelo menos x segundos” em tal estado antes que possa trocar para outro. Ela demonstra ainda o uso de variáveis mais complexas e seções críticas em ações.

Mais informações e exemplos podem ser encontrados no crate (SILVA, 2019c).

6.3 Macros Procedurais

Adicionar novas funcionalidades à linguagens de programação é, normalmente, um processo penoso e burocrático (RAFKIND, 2013). Ao mesmo tempo, já que as linguagens de propósito geral procuram satisfazer os casos mais comuns de abstração, elas nem sempre atendem às necessidades específicas de um domínio. Reconhecer que padrões particulares existem e que construções sintáticas apropriadas tornam os programas mais confiáveis (*dependable*), abre caminho para a adoção de mecanismos de abstração sintática (DYBVIG, 2007). Essas extensões empoderam os usuários a implementar suas próprias construções sem, necessariamente, passar por um comitê formal ou algo do gênero (RAFKIND, 2013).

Em linguagens com sintaxe homogênea, como Lisp (*S-expressions*), a metaprogramação é um processo natural; sempre foi, aliás, um dos diferenciais dessa família de dialetos. Os projetistas das demais linguagens buscam técnicas capazes de emular o efeito encontrado em Lisp, sem incorrer nos problemas enfrentados com *macros* pré processadas, como o sistema de *'defines'* em C.

Enfrenta-se uma série de desafios ao implementar um mecanismo de abstração sintática para linguagens infixas com delimitadores implícitos (*Algol-like*) (RAFKIND, 2013). Não é simples alterar a tecnologia de análise (*parser*) para suportar o uso de *macros*. Além disso, a sintaxe infixa cria problemas adicionais para a escrita de meta funções (RAFKIND, 2013). As *macros* procedurais em Rust, (RUST LANGUAGE, 2019j; CRICHTON, 2018), se oferecem como uma abordagem interessante para esse tema. Elas dão a impressão de "rodar na representação do programa", como *macros* em Lisp, embora realizem o processamento em texto (*tokens*).

A *macro* expansão ocorre durante a compilação da aplicação consumidora, antes de sua análise sintática. Os códigos expandidos e os criados a mão seguem juntos pelo restante do fluxo de compilação. Ambos estão estão sujeitos às regras sintáticas e semânticas, logo, as *macros* devem sempre resultar em código Rust válido.

Em suas primeiras versões, as *macros* operavam sobre a árvore sintática (AST), como nas linguagens homogêneas. Entretanto, como a sintaxe da linguagem Rust não fixa, mudanças na linguagem, até mesmos as retrocompatíveis, podiam resultar em quebras no sistema de metaprogramação. Percebeu-se que o caminho passava pelo uso de *macros* em contexto léxico. Em termos leigos, uma *macro* procedural é uma função que usa um "texto" para produzir outro, em tempo de

compilação; sua primeira definição formal aconteceu na RFC 1566, (VARIOUS CONTRIBUTORS, 2016). Todavia, o processo de estabilização foi longo e complicado, elas só se tornaram '`Stable`' no final de 2018. São, inclusive, uma das principais funcionalidades ressaltadas na edição Rust-2018 (RUST LANGUAGE, 2018b).

Melhores ferramentas de diagnóstico são providas em *macros* procedurais; o tratamento de erros em `macros_rules` é simplório, pouca informação é dada a respeito do real erro. As funções de metaprogramação são mais aptas a trabalhar com desenvolvimento conduzido por testes. Outro diferencial, nem sempre positivo, é o fato de requerem um *crate* próprio. Como as *macros* procedurais influenciam o fluxo de compilação, o *Cargo* precisa saber de antemão que não se trata de um programa convencional. No arquivo `Cargo.toml` deve haver a indicação de que esse pacote contém algum tipo de extensão sintática (`proc-macro = true`).

As *derive-like macros* estendem a funcionalidade do atributo `#[derive]`. Com eles é possível definir comportamentos "auto deriváveis" personalizados, um exemplo é serialização/deserialização de `structs` e `enums` provida pelo *crate* (SERDE-RS, 2019a). *Atribute-like macros* definem novos atributos, os quais podem ser anexado a um item. Com eles é possível, por exemplo, construir códigos semanticamente muito diferentes do tradicional. A aplicação descreve a lógica de negócio e informa "o que vai em cada lugar", através de anotações. Esse padrão é exaustivamente usado na *framework* `Rocket` (SERGIO BENITEZ, 2018). A DSL *state-machines,* apresentada como contribuição nesse texto, utiliza-se da terceira vertente *function-like.* Ela é, usualmente, conhecida como uma versão mais poderosa dos `macro_rules`.

A codificação de *macros* procedurais é um tópico avançado e relativamente recente. A documentação a respeito do tema ainda está em passos iniciais. As três vertentes se diferenciam construtivamente, com particularidades nos argumentos de entrada, informações visíveis e saídas automatizadas internamente. O restante do texto é direcionado à faceta baseada em funções, as técnicas aplicadas a seguir estão de acordo com as melhores práticas para o presente tempo.

Um objetivo importante a não perder de vista ao expandir uma linguagem é garantir que suas propriedades léxicas são mantidas. Esse fenômeno é conhecido como higiene (*hygiene*), ele busca assegurar que as novas ligações não irão conflitar com as que já estão na árvore sintática. As *macros* declarativas são higiênicas, já as procedurais não. Até o presente tempo, essa funcionalidade é instável e restrita ao canal *Nightly* (VARIOUS CONTRIBUTORS, 2018). Cabe ao programador

assegurar que os elementos usados internamente são válidos quando anexados à aplicação.

Uma *macro* procedural recebe `TokenStreams` como entrada e produz uma `TokenStream` como saída. Os *token*s são a menor unidade léxicas de uma linguagem, como identificadores, literais e pontuações. As `TokenStreams` são iteradores sobre uma árvore de *tokens* (`TokenTree`), vide Listagem 6.13. Elas vêm do *crate* `proc-macro` e são diretamente providas pelo compilador. Para contornar a dificuldade de trabalhar diretamente com essa sequência de símbolos, o primeiro passo é "parseá-los", ou seja, transformá-los em componentes Rust. A saída pressupõe o processo contrário, nela as expressões Rust são convertidas para seus símbolos equivalente, como mostrado em 6.13.

Listagem 6.13: 'TokenStream' é um referência a um vetor de 'tokens'. Sendo esses uma das variações do 'enum' 'TokenTree'.

```
// Nao eh uma representacao literal da estrutura interna.
// Abstratamente as APIs se assemelham, cabendo a aproximaÃ§Ã£o.
type TokenStream = Rc<Vec<TokenTree>>;

enum TokenTree {
    Group(Group),
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
}

// Como representar ‘let var = 42;‘, em termos de sua
// ‘TokenStream‘ equivalente.
TokenStream::from_iter(vec![
    TokenTree::from(Ident::new(‘‘let‘‘, span)),
    TokenTree::from(Ident::new(‘‘var‘‘, span)),
    TokenTree::from(Punct::new(‘‘=‘‘, Spacing::Alone)),
    TokenTree::from(Literal::u32_unsuffixed(42)),
    TokenTree::from(Punct::new(‘‘;‘‘, Spacing::Alone)),
])
```

Os *crates* `syn` e `quote`, simplificam os processos descritos acima, auxiliando na construção de *macros* procedurais. Além disso, como possuem forte ligação com estruturas internas ao compilador, tornam mais rápido o processo de expansão sintática.

`Syn` é uma biblioteca de *parser* criada para facilitar a transformação de uma cadeia de *tokens* em uma árvore sintática (TOLNAY, 2019b). Com ela o *parser* é construído em torno de funções analisadoras recursivas. Cada nó da AST definido pelo `syn` é analisável individualmente e pode ser usado como um bloco de construção para sintaxes personalizadas. A cada *token* disponível está associado um `Span` que rastreia sua linha e a coluna no código de origem.

O *crate* `Quote` (TOLNAY, 2019a) provém a *macro* `quote`. Seu objetivo é transformar árvores

sintáticas em *tokens* de um código fonte, evitando o processo manual mostrando na Listagem 6.13. Por trás dele está a ideia de "quase-citar" (HANSON, 2014), isto é, escrever *templates* de código como se estivesse escrevendo dados.

A função `sm`, através do atributo `#[proc_macro]`, define a base da linguagem de construção de máquinas de estado, isto é, a *macro* `sm!`. Os símbolos de entrada (`input`) são transformados na estrutura 'Machines', através do procedimento `parse_macro_input`. Ao final, a rotina retorna as "máquinas" (e seus comportamentos) em forma de *tokens* (`quote!(#machines).into()`).

Listagem 6.14: Definição do macro procedural 'sm'. Parseamento do 'tokens' de entrada gerando 'tokens' de saída.

```
1  use proc_macro::TokenStream;
2  use quote::quote;
3  use syn::parse_macro_input;
4
5  /// Definicao da `macro` procedural que vai gerar a `State Machine`.
6  #[proc_macro]
7  pub fn sm(input: TokenStream) -> TokenStream {
8      // Analise da `TokenStream` de entrada.
9      // `Syn`.
10     let machines: Machines = parse_macro_input!(input as Machines);
11
12     // Producao de uma cadeia de caracteres de saida.
13     // `Quote`.
14     quote!(#machines).into()
15 }
```

Observe que o sistema de tipos irá garantir que o fluxo ocorre conforme esperado. Para ser uma `#[proc_macro]` a rotina deve, impreterivelmente, receber uma `TokenStream` retornar uma `TokenStream`. Loops infinitos resultam em erro de compilação. `Panics` emitidos durante a construção são capturados pelo compilador e transformados em um erro de compilação.

O trabalho do *parser* é transformar o Código 6.8 em uma coleção de 'máquinas', ou, em termos técnicos, um vetor de 'Machine'. Sintaticamente, as máquinas estado são criadas descrevendo suas transições, estados iniciais e variáveis de controle. O elemento Rust equivalente deve possuir a mesma disposição. A estrutura 'Machine' foi criada para tal propósito (linha 1, Listagem 6.15). O campo 'name' é uma `Syn::Ident`, ou seja, uma palavra Rust válida (SERDE-RS, 2019b). As demais instâncias são estruturas Rust, analogamente, criadas para representar os componentes não atômicos.

Listagem 6.15: Definição de uma 'machine' para a linguagem de domínio específico.

```
1  pub struct Machine {
2      pub name: Ident,
```

```
    pub initial_states: InitialStates,
    pub transitions: Transitions,
    pub guard_resources: Guards,
    pub action_resources: Actions,
}

impl Parse for Machine {
    /// example machine tokens:
    ///
    /// ```text
    /// TurnStile {
    ///     GuardsResources { ... }
    ///     ActionResources { ... }
    ///     InitialStates { ... }
    ///
    ///     Push { ... }
    ///     Coin { ... }
    /// }
    /// ```
    ///
    fn parse(input: ParseStream<'_>) -> Result<Self> {
        // `TurnStile { ... }`
        //  ^^^^^^^^^
        let name: Ident = input.parse()?;

        // `TurnStile { ... }`
        //             ^^^
        let block_machine;
        braced!(block_machine in input);

        // `guard_resources`, `action_resources` e `initial_states`.
        ...;

        // `Push { ... }`
        //  ^^^^^^^^^^^^
        let transitions = Transitions::parse(&block_machine)?;

        Ok(Machine {
            name,
            initial_states,
            transitions,
            guard_resources,
            action_resources,
        })
    }
}
```

O processo de análise em camadas recursivas inicia no nível mais externo. O objetivo do *parser* da estrutura 'Machines' é operar sobre o código fonte, separando-o em "máquinas" válidas. Enquanto houver símbolos de entrada eles serão transmutados em 'Machine'. Se o resultado da conversão for verdadeiro (Ok(Machine{ ... }), 6.15), a 'machine' recebida é inserida no vetor 'machines'. A mesma sequência de passos acima descrita pode ser vista no código 6.16. Note que analisar (*parser*) um elemento é implementar-lhe o *trait* Parse.

Listagem 6.16: 'Machines' com seus respectivos parser e tokenizador.

```
...;
use syn::parse::{Parse, ParseStream, ...};

```

```
pub struct Machines(pub Vec<Machine>);

impl Parse for Machines {
    /// example machines tokens:
    ///
    /// ```text
    /// TurnStile { ... }
    /// Lock { ... }
    /// MyStateMachine { ... }
    /// ```
    ///
    fn parse(input: ParseStream<'_>) -> Result<Self> {
        let mut machines: Vec<Machine> = Vec::new();

        while !input.is_empty() {
            // `TurnStile { ... }`
            //  ^^^^^^^^^^^^^^^^^
            let machine = Machine::parse(input)?;
            machines.push(machine);
        }

        Ok(Machines(machines))
    }
}
```

Uma transição é composta por: um evento, um estado inicial e um estado final; `<evento>{inicio => fim}`. Tal comportamento foi traduzido para a estrutura 'Transition'. Note na Listagem 6.17 que, a função 'parse' correspondente implementa a separação dos *tokens* recebidos no padrão desejado. Espera-se um 'Event' seguido por um bloco. Interno ao 'block_transition', têm-se os estados envolvidos separados pelo símbolo '=>'. Todas as transições válidas são acumuladas no vetor 'transitions'.

Listagem 6.17: Tokenização das transições. Implementado o 'trait event' com suas respectivas particularidades.

```
pub struct Transitions(pub Vec<Transition>);

impl Parse for Transitions {
    fn parse(input: ParseStream<'_>) -> Result<Self> {
        let mut transitions: Vec<Transition> = Vec::new();
        while !input.is_empty() {
            // 'Coin { Locked => Unlocked }
            // ^^^^
            let event = Event::parse(input)?;

            // 'Coin { Locked => Unlocked }
            //         ^^^^^^^^^^^^^^^^^
            let block_transition;
            braced!(block_transition in input);

            let from = State::parse(&block_transition)?;

            let _: Token![=>] = block_transition.parse()?;

            let to = State::parse(&block_transition)?;

            transitions.push(Transition { event, from, to });
        }
```

```
24
25          Ok(Transitions(transitions))
26      }
27  }
28
29  pub struct Transition {
30      pub event: Event,
31      pub from: State,
32      pub to: State,
33  }
```

'States' e 'Events' são nomes arbitrários que, abstratamente, nomeiam estados e transições. Lexicalmente, eles são 'Syn::Ident', identificadores Rust. A Listagem 6.18 trás a representação de um 'estado', um 'evento' é interpretado de forma similar, por isso, foi omitidos desse texto.

Listagem 6.18: Representação de um 'state'.

```
1   pub struct State {
2       pub name: Ident,
3   }
4
5   impl Parse for State {
6       /// example state tokens:
7       ///
8       /// ```text
9       /// Locked
10      /// ```
11      ///
12      fn parse(input: ParseStream<'_>) -> Result<Self> {
13          let name = input.parse()?;
14
15          Ok(State { name })
16      }
17  }
```

Os componentes remanescentes 'InitialStates', 'Guards' e 'Actions', surgem de procedimentos análogos ao exibidos acima e não serão tratados explicitamente. Desde ponto em diante, o trabalho de análise está concluído e o vetor 'machines' contém todas a informações necessárias. A saída da *macro* surgirá a partir de manipulações procedurais em cima dele. Para aplicação em questão, o código resultante da expansão equivale aos mostrados na Subseção 6.2.4.

Expressividade é um dos valores centrais da DSL criada, assim sendo, nem todas as informações necessárias para construir a saída estão disponíveis diretamente. Os possíveis estados da máquina, por exemplo, são descobertos ao transversar uma 'machine' à procura dos estados de *reset* e dos pontos iniciais e finais das transições. O método 'states' (Listagem 6.19) abstrai tal operação. Assim, o vetor 'states' resultante contém, unicamente, todos os estados definidos pelo usuário.

Listagem 6.19: Definição do método 'states'.

```
impl Machine {
    fn states(&self) -> States {
        let mut states: Vec<State> = Vec::new();

        for t in &self.transitions.0 {
            if !states.iter().any(|s| s.name == t.from.name) {
                states.push(t.from.clone());
            }

            if !states.iter().any(|s| s.name == t.to.name) {
                states.push(t.to.clone());
            }
        }

        for i in &self.initial_states.0 {
            if !states.iter().any(|s| s.name == i.name) {
                states.push(State {
                    name: i.name.clone(),
                });
            }
        }

        States(states)
    }

    // Demais metodos.
    ...;
}
```

Um "estado", em baixo nível, é uma `struct` que implementa o traço `State`. Idiomaticamente, traduzir um conceito em sua implementação equivalente, é o mesmo que implementar-lhe o comportamente `ToTokens`. A função `to_tokens` deve, convenientemente, expandir a árvore de *tokens* resultante com os procedimentos relevantes. Através de *quasiquote,* a Listagem 6.20 elabora um modelo capaz de efetuar tal operação. Ao preceder uma expressão como caracter '`#`' informa-se ao compilador que ela deve ser avaliada (similar ao `comma` em Lisp). Assim, a sentença `#name` dará lugar ao nome do estado correspondente. Cabe ressaltar que o operador "avaliação" chama a transformação `to_tokens` do elemento que ele precede que, para o caso em questão, é o próprio '`name`' já que ele é uma '`Syn::Ident`'. As demais expressões são entregues de forma literal ao compilador, criando exatamente a saída desejada.

Listagem 6.20: Tokenização dos 'estados'.

```
impl ToTokens for State {
    fn to_tokens(&self, tokens: &mut TokenStream) {
        let name = &self.name;

        tokens.extend(
            quote! {
            pub struct #name;
            impl State for #name {}
            }
        );
    }
```

```
}
```

A função `to_tokens` de uma 'machine', apresentada na Listagem 6.21, controla o principal processo de expansão da DSL. As variáveis 'states', 'events', 'machine_eval' e assim por diante, abstraem operações de cada estrutura, organizando o processo de saída. A representação "enumerada", por exemplo, surge a partir da expansão da 'machine_enum'. O comportamento ToTokens (Listagem 6.21) de uma MachineEnum produz o Enum 'Variant' e as devidas conversões as_enum.

Listagem 6.21: Tokenização das 'machines'.

```
impl<'a> ToTokens for MachineEnum<'a> {
    fn to_tokens(&self, tokens: &mut TokenStream) {
        let mut variants = Vec::new();
        let mut states = Vec::new();
        let mut events = Vec::new();

        for s in &self.machine.initial_states.0 {
            let name = s.name.clone();
            let none = parse_quote! { NoneEvent };
            let variant = Ident::new(&format!("Initial{}", name),
                                     Span::call_site());

            variants.push(variant);
            states.push(name);
            events.push(none);
        }

        for t in &self.machine.transitions.0 {
            let state = t.to.name.clone();
            let event = t.event.name.clone();
            let variant = Ident::new(&format!("{}By{}", state, event),
                                     Span::call_site());

            if variants.contains(&variant) {
                continue;
            }

            variants.push(variant);
            states.push(state);
            events.push(event);
        }

        let variants = &variants;
        let states = &states;
        let events = &events;

        tokens.extend(quote!{
            pub enum Variant {
                #(#variants(Machine<#states, #events>)),*
            }

            #(
                impl AsEnum for Machine<#states, #events> {
                    type Enum = Variant;

                    fn as_enum(self) -> Self::Enum {
                        Variant::#variants(self)
                    }
```

```
                }
            )*
        });
    }
}

impl ToTokens for Machine {
    fn to_tokens(&self, tokens: &mut TokenStream) {
        let name = &self.name;
        let initial_states = &self.initial_states;
        let states = &self.states();
        let events = &self.events();
        let machine_enum = MachineEnum { machine: &self };
        let machine_eval = MachineEval { machine: &self };
        let transitions = &self.transitions;
        let guard_resources = &self.guard_resources;
        let action_resources = &self.action_resources;

        let Events(inside_event) = events;
        let mut use_events = Vec::new();

        for e in inside_event {
            let event_name = e.name.clone();
            use_events.push(quote!(use crate::#name::#event_name;));
        }

        tokens.extend(quote! {
            mod #name {
                pub struct Machine<S: State, E: Event>(S, Option<E>);

                impl<S: State, E: Event> M for Machine<S, E> {
                    type State = S;
                    type Event = E;

                    fn state(&self) -> Self::State {
                        self.0.clone()
                    }

                    fn trigger(&self) -> Option<Self::Event> {
                        self.1.clone()
                    }
                }

                impl<S: InitialState> Initializer<S> for Machine<S, NoneEvent>{
                    type Machine = Machine<S, NoneEvent>;

                    fn new(state: S) -> Self::Machine {
                        Machine(state, Option::None)
                    }
                }

                #states
                #initial_states
                #events
                #machine_enum
                #transitions

                pub trait ValidEvent {
                    fn is_enabled(#guard_resources) -> bool;
                    fn action(#action_resources);
                }
                pub trait MachineEvaluation {
                    fn eval_machine(&mut self, #guard_resources
                                    #action_resources) -> Result<(), ()>;
                }

```

```
115                 fn bool_to_u8(b: bool) -> u8 {
116                     if b { 1 } else { 0 }
117                 }
118
119                 #machine_eval
120             }
121         });
122     }
123 }
```

A implementação completa está disponível no repositório (SILVA, 2019e), sob a forma de um *crate*. Todos os *parser* e *tokenizadores* estão acompanhados de funções de teste unitário, servindo também como documentação dos comportamentos esperados. Esse pacote é completamente funcional e serviu como base para a construção da aplicação 6.12.

6.4 Considerações Finais

É importante notar que as técnicas empregadas ao longo desse capítulo se diferem profundamente, mas cada uma a seu modo, desempenha o que dela se espera: ajuda a controlar a complexidade enfrentada durante a codificação para sistemas embarcados. Os modelos propostos não são excludentes, foram apresentados (e usados) separadamente visando a didática do texto.

No que tange o sistema de construção de máquinas de estado, é possível evoluir com relação às análises estáticas efetuadas. Até aqui, as abstrações visam assegurar a confiabilidade e robustez das SMs criadas e, certificam que:

- apenas os estados iniciais informados estão disponíveis no método `new`;
- há apenas um estado corrente;
- somente as transições informadas estão disponíveis, com ações e predicados guarda explícitos e claros.

Por padrão, todas as máquinas criadas são recíprocas, ou seja, quando nenhuma transição está habilitada, o estado atual é mantido. Também por design, apenas transições determinísticas são efetuadas. As demais retornam um `Err` e cabe a aplicação a escolha do próximo estado. Com um '`unwrap`', idiomaticamente, transições não determinísticas resultam erros.

Trabalhos futuros pretendem explorar a natureza estática da DSL para extrair mais informações a respeito do sistema. Análises mais especulativas podem, por exemplo, transformar a exis-

tência de *deadlock* em erros (ou *warnings*) de compilação. Além disso, a sintaxe empregada na linguagem de aplicação específica 'SM' não passou por análises de usabilidade e também pode ser melhorada.

Ao fim desse capítulo, se encerra também as contribuições técnicas desse trabalho. O próximo capítulo fará o fechamento da dissertação e demais sugestões de trabalhos futuros.

Capítulo 7

Conclusões

A eficiência é, historicamente, o principal timoneiro das decisões técnicas durante o desenvolvimento de softwares embarcados. Entretanto, os recentes avanços tecnológicos e a profunda convergência entre humanos e máquinas, prescrevem uma mudança de paradigma. Confiamos a essas aplicações o controle de diversas atividades críticas, portanto, a performance não deve ser o único atributo a nortear desenvolvimento de um *firmware*. Visando a segurança das soluções geradas, esse trabalho defende que: robustez, integridade, estabilidade e resiliência precisam ser protagonistas importantes nas escolhas técnicas.

A linguagem C sempre enalteceu a simplicidade, cabendo ao programador se responsabilizar por todas as suas ações. Já a linguagem Rust nasceu com objetivos mais audaciosos, deseja aliar robustez, confiabilidade e expressividade com o desempenho esperado para um programa de baixo nível. Com uma mentalidade aberta e um código de conduta (RUST LANGUAGE, 2019c) que exalta a curiosidade e pesquisas profundas, Rust tem atraído a atenção de pesquisadores (e profissionais) envolvidos na codificação de sistema (*system programming*), em particular, a construção de softwares embarcados.

Rust é reconhecidamente uma linguagem de programação difícil. Conceitos teóricos sutis têm forte influência na linguagem e é muito fácil passar horas "brigando com o *borrow checker*". Além disso, enquanto os tópicos iniciais são brilhantemente apresentados na documentação oficial, temas avançados demandam a interpretação de questões (*issues*) não estruturadas no repositório da linguagem, (VARIOUS CONTRIBUTORS, 2019).

Por muito tempo, a restrição ao canal *Nightly* (sem garantias de retrocompatibilidade), notabilizou-

se em instabilidades no ecossistema *Rust-Embedded*. Até este ano, 2019, as escassas documentações eram, completamente, descentralizadas (postagens em blogs) e nem sempre condiziam com a realidade atual. Foi muito difícil construir o primeiro "acendendo um led". Sem uma organização global e com funcionalidades se desenvolvendo em paralelo (nem sempre acompanhadas de suas devidas explicações), a criação de aplicações microcontroladas passava longe de ser uma atividade prazerosa. As oscilações enfrentadas nos últimos três anos foram o preço a pagar pelo crescimento exponencial.

A biblioteca de abstração de hardware apresentada nesse trabalho mostrou ter grande potencial para melhorar a confiabilidade das aplicações embarcadas. Porém, ainda está incompleta e não suporta todos os periféricos existentes no dispositivo. Além disso, o sentimento "*rusty*", proveniente das abstrações do `embedded-hal`, se difere do usualmente encontrado em sistemas codificados em C. Encontrar formas de transpor essa lacuna foi, ou melhor, ainda é uma das principais dificuldades encontradas ao longo dessa pesquisa.

Como citado ao longo do texto, a plataforma RTFM ainda está em pleno desenvolvimento. Fato esse que direcionou a forma como as extensões seriam desenvolvidas. A pouca estabilidade sintática e semântica (aconteceram três grandes mudanças durante essa pesquisa) condicionaram a criação de ferramentas o menos intrusivas possível. Como vantagem, elas não dependem de artifícios internos e podem ser facilmente portadas entre versões. Como desvantagem, não tem um sentimento tão idiomático quanto as funcionalidades próprias da *framework*.

Com as devidas ponderações, acredita-se os sistemas propostos são satisfatórios e cumprem o que prometeram ao auxilar no controle da complexidade nos softwares embarcados, a partir abstrações seguras e confiáveis. Trabalhos futuros e possíveis adequações podem tornar as ferramentas apresentadas ainda mais adaptadas a realidade. Há um longo caminho a percorrer, todavia, os resultados obtidos até aqui, indicam um próspero caminho a frente.

7.1 Trabalhos Futuros

Para dar continuidade ao desenvolvimento desse projeto, algumas funcionalidades devem ser refinadas e melhoradas. Mais periféricos precisam ser suportados pelo *HAL*, para que aplicações mais convincentes possam ser criadas. A falta de exemplos mais reais faz com que, em determina-

dos pontos, esse trabalho parece desconexo com a realidade, ponto a ser resolvido em um futuro próximo, tornando as contribuições propostas mais palpáveis.

Até aqui, a DSL de construção de máquinas de estados permite somente a criação máquinas *Mealy*. Para algumas situações reais, a descrição via máquina *Moore* é mais adequada ao problema. Uma segunda implementação com esse fim, está no radar como possíveis pontos de atuação no futuro. Além disso, os erros (*spams*) provenientes das *macros* procedurais não foram tratados com o devido cuidado, as mensagens de erro, até aqui, são pouco expressivas. Como já citado, a sintaxe e as análises efetuadas também pode evoluir em versões futuras.

A falta de dados contundentes, ressaltando os *trade-offs* envolvidos na troca de linguagem, também contribuem para a aparente dissociação com o mundo real. Tem-se consciência de que para atingir público, fora do ambiente acadêmico, é necessário apresentar comparações e *benchmarks* mais abrangentes. Comprovar a eficiência das soluções propostas é o principal ponto de atuação para trabalhos futuros.

Finalmente, as técnicas aqui descritas podem ser avaliadas ou adaptadas para outros tipo de dispositivos embarcados. Tendências como sistemas heterogêneos assimétricos são uma evolução natural para esse trabalho. Os primitivos Rust e o tratamento seguro da concorrência podem ser de grande valia nesse tipo de equipamento, abrindo um grande campo de atuação.

Bibliografia

ADA WORKING GROUP. **Ada Reference Manual**. [S.l.: s.n.], 15 jun. 2001. Disponível em: <https://www.adaic.org/resources/add_content/standards/95lrm/RM.pdf>. Acesso em: 14 mar. 2019.

AMAZON WEB SERVICES. **FreeRTOS**. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. 9 mar. 2019. Disponível em: <https://www.freertos.org/>. Acesso em: 9 mar. 2019.

AMEY, Peter. Correctness by Construction: Better Can Also Be Cheaper, p. 6, 2002.

ANA HOBDEN. **Pretty State Machine Patterns in Rust**. Hoverbear. 12 out. 2016. Disponível em: <https://hoverbear.org/2016/10/12/rust-state-machine-pattern/>. Acesso em: 15 abr. 2019.

APARICIO, Jorge. **ARMv6: use interrupt masking instead of "global"critical sections**. [S.l.: s.n.], 14 nov. 2018. original-date: 2017-03-05T05:28:17Z. Disponível em: <https://github.com/japaric/cortex-m-rtfm>. Acesso em: 25 mar. 2019.

_____. **Cortex-m-rtfm. RFC, interrupt binds.** [RFC] #[interrupt(binds = "..")] #128. original-date: 2017-03-05T05:28:17Z. 11 jan. 2019. Disponível em: <https://github.com/japaric/cortex-m-rtfm>. Acesso em: 23 mar. 2019.

_____. **Explicit ‘Context‘ parameter.** original-date: 2017-03-05T05:28:17Z. 20 mar. 2019. Disponível em: <https://github.com/japaric/cortex-m-rtfm>. Acesso em: 20 mar. 2019.

_____. **Heapless - Rust**. Crate heapless. 2019. Disponível em: <https://japaric.github.io/heapless/heapless/index.html>. Acesso em: 21 mar. 2019.

APARICIO, Jorge. **Implementing a static stack usage analysis tool - Embedded in Rust**. Stack Analysis. 13 mar. 2019. Disponível em: <https://blog.japaric.io/stack-analysis/>. Acesso em: 19 mar. 2019.

_____. **Invoking software tasks**. original-date: 2017-03-05T05:28:17Z. 25 jan. 2019. Disponível em: <https://github.com/japaric/cortex-m-rtfm>. Acesso em: 27 mar. 2019.

_____. **japaric - Overview**. GitHub. 2019. Disponível em: <https://github.com/japaric>. Acesso em: 17 mar. 2019.

_____. **nb - Rust**. 2018. Disponível em: <https://japaric.github.io/nb/nb/>. Acesso em: 4 abr. 2019.

_____. **Program static stack analysis.** [S.l.: s.n.], 21 mar. 2019. original-date: 2018-12-03T01:52:30Z. Disponível em: <https://github.com/japaric/cargo-call-stack>. Acesso em: 23 mar. 2019.

_____. **Rust Embedded development on stable**. GitHub. Fev. 2018. Disponível em: <https://github.com/rust-embedded/wg/issues/42>. Acesso em: 30 mar. 2019.

_____. **Shared access to resources (resources = [&FOO])**. original-date: 2017-03-05T05:28:17Z. 20 mar. 2019. Disponível em: <https://github.com/japaric/cortex-m-rtfm>. Acesso em: 20 mar. 2019.

_____. **SPSC - split**. [S.l.: s.n.], 3 mai. 2019. original-date: 2017-03-05T05:32:09Z. Disponível em: <https://github.com/japaric/heapless>. Acesso em: 6 mai. 2019.

_____. **Xargo in maintenance mode.** GitHub. Jan. 2018. Disponível em: <https://github.com/japaric/xargo/issues/193>. Acesso em: 30 mar. 2019.

ARM. Cortex-M0+ Devices Generic User Guide. **ARM**, p. 113, 2012.

AVIZ, Algirdas; LAPRIE, Jean-Claude; RANDELL, Brian. Fundamental Concepts of Dependability. **Proc. 3rd Information Survivability Workshop**, p. 21, 2000.

BAKER, T.P. A stack-based resource allocation policy for realtime processes. In: [1990] PROCEEDINGS 11TH REAL-TIME SYSTEMS SYMPOSIUM. **[1990] Proceedings 11th Real-Time Systems Symposium**. Lake Buena Vista, FL, USA: IEEE, 1990. p. 191–200. ISBN 978-0-8186-2112-3. DOI: 10.1109/REAL.1990.128747. Disponível em: <http://ieeexplore.ieee.org/document/128747/>. Acesso em: 11 mar. 2019.

BALDASSARI, François. **RustyPebble**. [S.l.: s.n.], 21 ago. 2014. original-date: 2014-08-16T21:30:38Z. Disponível em: <https://github.com/franc0is/RustyPebble>. Acesso em: 15 mar. 2019.

BATTY, Mark John. **The C11 and C++11 Concurrency Model**. 29 nov. 2014. Doctor of Philosophy – University of Cambridge.

BERRY, G. Real time programming: Special purpose or general purpose languages. **IFIP World Computer Congress**, 1989.

BERRY, Gérard. Proof, Language, and Interaction. In: PLOTKIN, Gordon; STIRLING, Colin; TOFTE, Mads (Ed.). Cambridge, MA, USA: MIT Press, 2000. p. 425–454. ISBN 978-0-262-16188-6. Disponível em: <http://dl.acm.org/citation.cfm?id=345868.345908>. Acesso em: 30 abr. 2019.

BLANDY, Jim. **Why Rust? Trustworthy, Concurrent Systems Programming**. 1. ed. United States of America: OReilly Media, 2015. ISBN 978-1-4919-2730-4.

BOYAPATI, Chandrasekhar; LEE, Robert; RINARD, Martin. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. **OOPSLA: ACM Conference on Object-Oriented Programming, Systems, Languages and Applications**, p. 20, nov. 2002. DOI: 10.1145/582438.582440.

BROOKS, Frederick. No Silver Bullet Essence and Accident in Software Engineering, p. 16, 1986.

BURNS, Alan; WELLINGS, Andy. **Real-time systems and programming languages: Ada, Real-Time Java and C/Real-Time POSIX**. 4th edition. Harlow, England London New York Boston San Francisco Toronto Sydney: Addison-Wesley, 2009. 602 p. (International computer science series). OCLC: 298597735. ISBN 978-0-321-41745-9.

BUSCHMANN, Frank et al. **Pattern-Oriented Sotfware Architecture**. Chichester ; New York: Wiley, 1996. 305 p. ISBN 978-0-471-96174-1 978-0-471-95889-5.

BUTTAZZO, Giorgio C. **Hard real-time computing systems: predictable scheduling algorithms and applications**. 3rd ed. New York: Springer, 2011. 521 p. (Real-time systems series). OCLC: ocn741541202. ISBN 978-1-4614-0675-4 978-1-4614-0676-1. DOI: 10.1007/978-1-4614-0676-1_3.

CADAR, Cristian; DUNBAR, Daniel; ENGLER, Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, p. 16, 8 dez. 2008.

CHESS. **Precision Timed (PRET) Machines**. Precision Timed (PRET) Machines. 2015. Disponível em: <https://ptolemy.berkeley.edu/projects/chess/pret/>.

CHURCH, Alonzo. Review: Edward F. Moore, Gedanken-Experiments on Sequential Machines. **Journal of Symbolic Logic**, v. 23, n. 1, p. 60–60, 1958. DOI: 10.2307/2964500.

CONWAY, Melvin E. HOW DO COMMITTEES INVENT?, p. 4, 1968.

CRICHTON, Alex. **Procedural Macros in Rust 2018**. Procedural Macros in Rust 2018 | Rust Blog. 21 dez. 2018. Disponível em: <https://blog.rust-lang.org/2018/12/21/Procedural-Macros-in-Rust-2018.html>. Acesso em: 22 abr. 2019.

DYBVIG, R Kent. Syntactic Abstraction: The syntax-case expander. **Indiana University**, p. 23, jun. 2007.

ELLIS, Ferris. **Math with distances in Rust: safety and correctness across units**. 8 mar. 2017. Disponível em: <https://ferrisellis.com/posts/rust-implementing-units-for-types/>. Acesso em: 4 mai. 2019.

EMBEDDED RUST WORKING GROUP. **Rust Embedded**. GitHub. 2019. Disponível em: <https://github.com/rust-embedded>. Acesso em: 17 mar. 2019.

FAIRBANKS, George. **Just Enough Software Architecture: A Risk-Driven Approach**. Boulder, Colorado: Marshall & Brainerd, ago. 2010. ISBN 978-0-9846181-0-1.

FARIA, Jose Miguel Sampaio. **Formal Development of Solutions for Real-Time Operating Systems with TLA + /TLC**. Jan. 2008. 139 p. Master degree – Universidade do Porto, Porto. Disponível em: <https://repositorio-aberto.up.pt/bitstream/10216/11466/2/Texto%20integral.pdf>. Acesso em: 13 mar. 2019.

FOWLER, Martin. **GOTO 2017 - The Many Meanings of Event-Driven Architecture**. 11 mai. 2017. Disponível em: <https://www.youtube.com/watch?v=STKCRSUsyP0&t=304s>. Acesso em: 21 mai. 2019.

GAMMA, Erich et al. Design Patterns : Elements of Reusable Object-Oriented Software, p. 431, ago. 1997.

GLISTVAIN, Roman; ABOELAZE, Mokhtar. Romantiki OS, A single stack Multitasking Operating System for Resource Limited Embedded Devices. **The 7th international conference on informatics and systems (INFOS)**, p. 8, 2010.

GODSE, A. P.; GODSE, D. A. **Advanced Microprocessors & Microcontrollers**. 1. ed. Pune, Índia: Technical Publications Pune, 2008. ISBN 978-81-8431-349-9.

GOLANG.ORG. **The Go Programming Language**. 2019. Disponível em: <https://golang.org/>. Acesso em: 5 jun. 2019.

GOODRICH, Elliot. **Memory optimal Small String Optimization implementation for C++: elliotgoodrich/SSO-23**. [S.l.: s.n.], 20 fev. 2019. original-date: 2014-12-03T00:03:53Z. Disponível em: <https://github.com/elliotgoodrich/SSO-23>. Acesso em: 10 mar. 2019.

HALBWACHS, Nicolas. Synchronous Programming of Reactive Systems, p. 182, 1993. DOI: 10.1007/978-1-4757-2231-4.

_____. The synchronous data-?ow language Lustre. **Proceedings of the IEEE**, v. 79, n. 9, p. 7, 1991. DOI: 10.1109/5.97300. Acesso em: 30 abr. 2019.

HAMMOND, Kevin; MICHAELSON, Greg. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In: PFENNING, Frank; SMARAGDAKIS, Yannis (Ed.). **Generative Programming and Component Engineering**. Redigido por Gerhard Goos, Juris Hartmanis e Jan van Leeuwen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. v. 2830. p. 37–56. ISBN

978-3-540-20102-1 978-3-540-39815-8. DOI: 10.1007/978-3-540-39815-8_3. Disponível em: <http://link.springer.com/10.1007/978-3-540-39815-8_3>. Acesso em: 20 mar. 2019.

HANSON, Chris. Quoting. In: MIT/GNU Scheme Reference Manual. 1.105. ed. [S.l.]: Free Software Foundation, 5 mai. 2014.

HAREL, D; PNUELI, A. On the development of reactive systems. In: LOGIC and Models of Concurrent Systems, NATO Advanced Study Institute on Logics and Models for Veri- cation and Speci cation of Concurrent Systems . [S.l.]: Springer Verlag, 1985. DOI: 10.1007/978-3-642-82453-1_17.

HAVERBEKE, Marijn. **The Rust That Could Have Been**. 25 jan. 2017. Disponível em: <https://www.youtube.com/watch?v=olbTX95hdbg&list=LLN-W7QqXSRcMl7ppltTM_aA&index=7&t=0s>. Acesso em: 8 jun. 2019.

HE, Kaifei; JIN, Man; KARLTUN, Johan. Cyber-Physical System for maintenance in industry 4.0, p. 64, 11 nov. 2016.

JUNG, Ralf et al. RustBelt: securing the foundations of the rust programming language. **Proceedings of the ACM on Programming Languages**, v. 2, p. 1–34, POPL 27 dez. 2017. ISSN 24751421. DOI: 10.1145/3158154. Disponível em: <http://dl.acm.org/citation.cfm?doid=3177123.3158154>. Acesso em: 10 mar. 2019.

K M BHURCHANDI; A K RAY. **Advanced microprocessors and peripherals**. 3. ed. New Delhi, India: Tata McGraw Hill Education Private Limited, 2013. ISBN 978-1-25-9006613-5.

KIRILL. **Asynchronous programming.** Blog | iamluminousmen. 3 mar. 2019. Disponível em: <//luminousmen.com/post/asynchronous-programming-python3.5>. Acesso em: 22 mai. 2019.

KOHLBECKER, E. E.; WAND, M. Macro-by-example: Deriving syntactic transformations from their specifications. In: THE 14TH ACM SIGACT-SIGPLAN SYMPOSIUM. **Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87**. Munich, West Germany: ACM Press, 1987. p. 77–84. ISBN 978-0-89791-215-0. DOI: 10.1145/41625.41632. Disponível em: <http://portal.acm.org/citation.cfm?doid=41625.41632>. Acesso em: 17 mar. 2019.

LAMPORT, Leslie. Specifying Concurrent Program Modules. **ACM Transactions on Programming Languages and Systems**, v. 5, n. 2, p. 190–222, 1 abr. 1983. ISSN 01640925. DOI: 10.1145/69624.357207. Disponível em: <http://portal.acm.org/citation.cfm?doid=69624.357207>. Acesso em: 27 mar. 2019.

The Specification Language TLA+. In: LAMPORT, Leslie (Ed.). **Logics of Specification Languages**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 401–451. ISBN 978-3-540-74106-0. DOI: 10.1007/978-3-540-74107-7_8. Disponível em: <http://link.springer.com/10.1007/978-3-540-74107-7_8>. Acesso em: 13 mar. 2019.

LASI HEINER et al. Industry 4.0. **Business & Information Systems Engineering: The International Journal of WIRTSCHAFTSINFORMATIK**, v. 6, p. 239–242, 2014. DOI: 10.1007/s12599-014-0334-4. Disponível em: <https://EconPapers.repec.org/RePEc:spr:binfse:v:6:y:2014:i:4:p:239-242>.

LEE, Edward A. Cyber Physical Systems: Design Challenges. **Object Oriented Real- Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on**, p. 7, 6 mai. 2008. DOI: 10.1109/ISORC.2008.25.

_____. Embedded Software. In: ZELKOWITZ, M. (Ed.). **Advances in Computers**. London: Academic Press, 2002. v. 56. p. 34. DOI: 10.1016/S0065-2458(02)80004-3.

_____. Model-Driven Development - From Object-Oriented Design to Actor-Oriented Design. **Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation**, p. 7, 24 set. 2003.

_____. The Problem with Threads. **Technical Report No. UCB/EECS-2006-1**, 10 jan. 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html>.

_____. What Does 'Real Time' Mean. Simons Institute, 28 jun. 2016. Disponível em: <https://www.youtube.com/watch?v=G8aofqZxVZY>.

LEE, Edward Ashford; SESHIA, Sanjit Arunkumar. **Introduction to embedded systems: a cyber physical systems approach**. 1. ed., print. 1.08. Lulu: LeeSeshia.org, 2012. 495 p. OCLC: 915495730. ISBN 978-0-557-70857-4.

LEE, Jay; BAGHERI, Behrad; KAO, Hung-An. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. **Manufacturing Letters**, v. 3, p. 18–23, jan. 2015. ISSN 22138463. DOI: 10.1016/j.mfglet.2014.12.001. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S221384631400025X>. Acesso em: 20 mar. 2019.

LEVESON, Nancy. **Engineering a safer world: systems thinking applied to safety**. Cambridge, Mass: MIT Press, 2011. 534 p. (Engineering systems). OCLC: ocn719429220. ISBN 978-0-262-01662-9. DOI: 10.7551/mitpress/8179.001.0001.

LEZOCHE, Mario; PANETTO, Hervé. Cyber-Physical Systems, a new formal paradigm to model redundancy and resiliency. **Enterprise Information Systems**, p. 1–22, 8 nov. 2018. ISSN 1751-7575, 1751-7583. DOI: 10.1080/17517575.2018.1536807. Disponível em: <https://www.tandfonline.com/doi/full/10.1080/17517575.2018.1536807>. Acesso em: 8 mar. 2019.

LI, Qing; YAO, Caroline. **Real-time concepts for embedded systems**. San Francisco, CA: CMP Books, 2003. 294 p. ISBN 978-1-57820-124-2. DOI: 10.1201/9781482280821.

LINDGREN, Per; FRESK, Emil et al. Abstract timers and their implementation onto the ARM Cortex-M family of MCUs. **ACM SIGBED Review**, v. 13, n. 1, p. 48–53, 25 mar. 2016. ISSN 15513688. DOI: 10.1145/2907972.2907979. Disponível em: <http://dl.acm.org/citation.cfm?doid=2907972.2907979>. Acesso em: 25 mar. 2019.

LINDGREN, Per; LINDNER, Marcus et al. RTFM-core: Language and implementation. In: 2015 IEEE 10TH CONFERENCE ON INDUSTRIAL ELECTRONICS AND APPLICATIONS (ICIEA). **2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)**. Auckland, New Zealand: IEEE, jun. 2015. p. 990–995. ISBN 978-1-4799-8389-6. DOI: 10.1109/ICIEA.2015.7334252. Disponível em: <http://ieeexplore.ieee.org/document/7334252/>. Acesso em: 8 mar. 2019.

LINDNER, Andreas. **Kernel support for MPU based memory protection in RTFM-core**. 23 jan. 2015. 61 p. Tese (Doutorado) – freie universität berlin.

LINDNER, Marcus; APARICIO, Jorge et al. Hardware-in-the-loop based WCET analysis with KLEE. In: 2018 IEEE 23RD INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION (ETFA). **2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)**. Turin: IEEE, set. 2018. p. 345–352. ISBN 978-1-5386-7108-5. DOI: 10.1109/ETFA.2018.8502510. Disponível em: <https://ieeexplore.ieee.org/document/8502510/>. Acesso em: 20 mar. 2019.

LINDNER, Marcus; APARICIUS, Jorge; LINDGREN, Per. No Panic! Verification of Rust Programs by Symbolic Execution. In: 2018 IEEE 16TH INTERNATIONAL CONFERENCE ON INDUSTRIAL INFORMATICS (INDIN). **2018 IEEE 16th International Conference on Industrial Informatics (INDIN)**. Porto: IEEE, jul. 2018. p. 108–114. ISBN 978-1-5386-4829-2. DOI: 10.1109/INDIN.2018.8471992. Disponível em: <https://ieeexplore.ieee.org/document/8471992/>. Acesso em: 1 abr. 2019.

LINDNER, Marcus; LINDNER, Andreas; LINDGREN, Per. RTFM-core: Course in Compiler Construction. In: THE WESE'14: WORKSHOP. **Proceedings of the WESE'14: Workshop on Embedded and Cyber-Physical Systems Education - WESE'14**. New Delhi, India: ACM Press, 2015. p. 1–9. ISBN 978-1-4503-3090-9. DOI: 10.1145/2829957.2829962. Disponível em: <http://dl.acm.org/citation.cfm?doid=2829957.2829962>. Acesso em: 11 mar. 2019.

_____. Safe tasks: Run time verification of the RTFM-lang model of computation. In: 2016 IEEE 21ST INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION (ETFA). **2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)**. Berlin, Germany: IEEE, set. 2016. p. 1–8. ISBN 978-1-5090-1314-2. DOI: 10.1109/ETFA.2016.7733550. Disponível em: <http://ieeexplore.ieee.org/document/7733550/>. Acesso em: 11 mar. 2019.

LLVM PROJECT. **LLD - The LLVM Linker lld 9 documentation**. 12 mar. 2019. Disponível em: <https://lld.llvm.org/>. Acesso em: 1 abr. 2019.

_____. **The LLVM Compiler Infrastructure Project**. 2019. Disponível em: <https://llvm.org/>. Acesso em: 9 mar. 2019.

LULEÃ. **Luleå University of Technology**. 2019. Disponível em: <https://www.ltu.se/?l=en>. Acesso em: 10 mar. 2019.

MANSFIELD, Tim. **Idiomatic coding**. Idiomatic coding · tim-hr/stuff Wiki. original-date: 2016-10-18T08:38:31Z. 18 out. 2016. Disponível em: <https://github.com/tim-hr/stuff>. Acesso em: 13 mar. 2019.

MARK ELENDT. **CppCon 2018: Dangling in French and English**. 19 nov. 2018. Disponível em: <https://www.youtube.com/watch?v=jieYLTcmTS0>. Acesso em: 10 mar. 2019.

MATTERN, Friedemann; FLOERKEMEIER, Christian. From the Internet of Computers to the Internet of Things. In: SACHS, Kai; PETROV, Ilia; GUERRERO, Pablo (Ed.). **From Active Data Management to Event-Based Systems and More**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. v. 6462. p. 242–259. ISBN 978-3-642-17225-0 978-3-642-17226-7. DOI: 10.1007/978-3-642-17226-7_15. Disponível em: <http://link.springer.com/10.1007/978-3-642-17226-7_15>. Acesso em: 9 mar. 2019.

MEALY, George H. **BSTJ 34: 5. September 1955: A Method for Synthesizing Sequential Circuits. (Mealy, George H.)** [S.l.: s.n.], set. 1955. DOI: 10.1002/j.1538-7305.1955.tb03788.x. Disponível em: <http://archive.org/details/bstj34-5-1045>. Acesso em: 9 abr. 2019.

MENTOR. **Nucleus RTOS**. Nucleus RTOS. 9 mar. 2019. Disponível em: <https://www.mentor.com/embedded-software/nucleus/>. Acesso em: 9 mar. 2019.

MERNIK, Marjan; HEERING, Jan; SLOANE, Anthony M. When and how to develop domain-specific languages. **ACM Computing Surveys**, v. 37, n. 4, p. 316–344, 1 dez. 2005. ISSN 03600300. DOI: 10.1145/1118890.1118892. Disponível em: <http://portal.acm.org/citation.cfm?doid=1118890.1118892>. Acesso em: 20 mar. 2019.

MERTZ, Jean. **sm 0.7.0**. sm 0.7.0 - Docs. 2019. Disponível em: <https://docs.rs/crate/sm/>. Acesso em: 15 abr. 2019.

MICHAEL, M M; SCOTT, M L. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms., p. 14, dez. 1995. DOI: 10.21236/ADA309412.

MIKHAJLOV, Leonid; SEKERINSKI, Emil. A study of the fragile base class problem. In: JUL, Eric (Ed.). **ECOOP98 Object-Oriented Programming**. Redigido por Gerhard Goos, Juris Hartmanis e Jan van Leeuwen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. v. 1445. p. 355–382. ISBN 978-3-540-64737-9 978-3-540-69064-1. DOI: 10.1007/BFb0054099. Disponível em: <http://link.springer.com/10.1007/BFb0054099>. Acesso em: 27 mar. 2019.

MILLER, George A. The Magical Number Seven, Plus or Minus Two Some Limits on Our Capacity for Processing Information. **Psychological Review**, v. 101, p. 343–352, 4 mai. 1955. DOI: 10.1037/0033-295X.101.2.343. Disponível em: <http://spider.apa.org/ftdocs/rev/1994/april/rev1012343.htm>.

MITCHELL, John C. On abstraction and the expressive power of programming languages. **Science of Computer Programming**, v. 21, p. 23, 1993. DOI: 10.1016/0167-6423(93)90004-9.

MOSELEY, Ben; MARKS, Peter. Out of the Tar Pit, p. 66, 6 fev. 2006.

NASH, Phil. **CppCon 2018: You're Not as Smart as You Think You Are**. 18 nov. 2018. Disponível em: <https://www.youtube.com/watch?v=fbkfH0IZW8g>. Acesso em: 21 mai. 2019.

NAUR, Peter; RANDELL, Brian. NATO SOFTWARE ENGINEERING CONFERENCE 1968, p. 136, 1968.

NGUYEN-DUC, Anh. The Impact of Software Complexity on Cost and Quality - A Comparative Analysis Between Open Source and Proprietary Software. **International Journal of Software Engineering & Applications**, v. 8, n. 2, p. 17–31, 31 mar. 2017. ISSN 09762221, 09759018. DOI: 10.5121/ijsea.2017.8202. Disponível em: <http://www.aircconline.com/ijsea/V8N2/8217ijsea02.pdf>. Acesso em: 19 mar. 2019.

NORVIG, Peter. Design Patterns in Dynamic Programming. [S.l.], 5 mai. 1996.

PARNAS, D L. On the Criteria To Be Used in Decomposing Systems into Modules. v. 15, n. 12, p. 6, 1972. DOI: 10.1145/361598.361623.

PATHAN, Risat Mahmud. Report for the Seminar Series on Software Failures, p. 14.

PIERCE, Benjamin C. **Types and programming languages**. Cambridge, Mass: MIT Press, 2002. 623 p. ISBN 978-0-262-16209-8.

PIKE, Rob. **OSCON 2010: Rob Pike, "Public Static Void- YouTube**. OSCON 2010: "Public Static Void". 9 mar. 2019. Disponível em: <https://www.youtube.com/watch?v=5kj5ApnhPAE>. Acesso em: 9 mar. 2019.

PRESSMAN, Roger S. **Software engineering: a practitioner's approach**. 5th ed. Boston, Mass: McGraw Hill, 2000. 860 p. (McGraw-Hill series in computer science). ISBN 978-0-07-365578-9.

PRESTON-WERNER, Tom. **Semantic Versioning 2.0.0**. Semantic Versioning. 2019. Disponível em: <https://semver.org/>. Acesso em: 31 mar. 2019.

QUANTUM LEAPS. **QP/C (QuantumPlataform in C)**. 2019. Disponível em: <http://www.state-machine.com/qpc/>. Acesso em: 26 mar. 2019.

RAFKIND, Jon. **Syntactic extension for languages with implicitly delimited and infix syntax**. Fev. 2013. Doctor of Philosophy – University of Utah, Utah.

RIVERA, Jorge Aparicio; LINDNER, Marcus; LINDGREN, Per. Heapless: Dynamic Data Structures without Dynamic Heap Allocator for Rust. In: 2018 IEEE 16TH INTERNATIONAL CONFERENCE ON INDUSTRIAL INFORMATICS (INDIN). **2018 IEEE 16th International Conference on Industrial Informatics (INDIN)**. Porto: IEEE, jul. 2018. p. 87–94. ISBN 978-1-5386-4829-2. DOI: 10.1109/INDIN.2018.8472097. Disponível em: <https://ieeexplore.ieee.org/document/8472097/>. Acesso em: 21 mar. 2019.

RUST EMBEDDED. **Awesome Embedded Rust.** awesome-embedded-rust. 2019. Disponível em: <https://github.com/rust-embedded/awesome-embedded-rust>. Acesso em: 31 mar. 2019.

_____. **Cortex-m**. [S.l.]: Rust Embedded, 14 mar. 2019. original-date: 2016-09-27T23:35:04Z. Disponível em: <https://github.com/rust-embedded/cortex-m>. Acesso em: 18 mar. 2019.

_____. **Cortex-m-rt**. [S.l.]: Rust Embedded, 15 mar. 2019. original-date: 2017-03-12T15:54:13Z. Disponível em: <https://github.com/rust-embedded/cortex-m-rt>. Acesso em: 18 mar. 2019.

RUST EMBEDDED. **Embedded-Hal**. [S.l.]: Rust Embedded, 16 mar. 2019. original-date: 2017-06-09T20:57:29Z. Disponível em: <https://github.com/rust-embedded/embedded-hal>. Acesso em: 18 mar. 2019.

_____. **Stable assembly operations**. GitHub. 2018. Disponível em: <https://github.com/rust-embedded/wg/issues/63>. Acesso em: 1 abr. 2019.

_____. **SVD2Rust**. [S.l.]: Rust Embedded, 18 mar. 2019. original-date: 2016-10-09T02:47:52Z. Disponível em: <https://github.com/rust-embedded/svd2rust>. Acesso em: 18 mar. 2019.

_____. **The Embedonomicon**. The Embedonomicon. 2019. Disponível em: <https://docs.rust-embedded.org/embedonomicon/>. Acesso em: 19 mar. 2019.

RUST LANGUAGE. **Advanced Types - The Rust Programming Language**. Advanced Types - The Rust Programming Language. 2019. Disponível em: <https://doc.rust-lang.org/1.30.0/book/2018-edition/ch19-04-advanced-types.html?highlight=diver#the-never-type-that-never-returns>. Acesso em: 17 mar. 2019.

_____. **Box**. 2019. Disponível em: <https://doc.rust-lang.org/std/boxed/struct.Box.html>. Acesso em: 6 jun. 2019.

_____. **Code of conduct**. Code of conduct. 2019. Disponível em: <https://www.rust-lang.org/policies/code-of-conduct>. Acesso em: 29 abr. 2019.

_____. **Comparing Performance: Loops vs. Iterators**. 2019. Disponível em: <https://doc.rust-lang.org/book/ch13-04-performance.html>. Acesso em: 31 mar. 2019.

_____. **Exotically Sized Types.** Exotically Sized Types. 2019. Disponível em: <https://doc.rust-lang.org/nomicon/exotic-sizes.html>. Acesso em: 15 mar. 2019.

_____. **FnOnce**. Trait std::ops::FnOnce. 2019. Disponível em: <https://doc.rust-lang.org/std/ops/trait.FnOnce.html>. Acesso em: 18 mar. 2019.

_____. **How Rust is Made and Nightly Rust**. 2019. Disponível em: <https://doc.rust-lang.org/book/appendix-07-nightly-rust.html?highlight=release,channels#choo-choo-release-channels-and-riding-the-trains>. Acesso em: 9 mar. 2019.

RUST LANGUAGE. **Marker - Primitive traits and types representing basic properties of types.** 2019. Disponível em: <https://doc.rust-lang.org/std/marker/index.html>. Acesso em: 5 abr. 2019.

_____. **Meet Safe and Unsafe**. 2018. Disponível em: <https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html>. Acesso em: 10 mar. 2019.

_____. **Object Oriented Programming Features of Rust**. 2019. Disponível em: <https://doc.rust-lang.org/book/ch17-01-what-is-oo.html?highlight=inheri#inheritance-as-a-type-system-and-as-code-sharing>. Acesso em: 27 mar. 2019.

_____. **Procedural Macros - The Rust Reference**. Abr. 2019. Disponível em: <https://doc.rust-lang.org/reference/procedural-macros.html>. Acesso em: 22 abr. 2019.

_____. **Rust Platform Support**. Rust Forge. 2019. Disponível em: <https://forge.rust-lang.org/platform-support.html>. Acesso em: 18 mar. 2019.

_____. **Rust programming language**. Rust language. 2019. Disponível em: <https://www.rust-lang.org/>. Acesso em: 9 mar. 2019.

_____. **The Edition Guide**. The Edition Guide. 2018. Disponível em: <https://doc.rust-lang.org/edition-guide/editions/index.html>. Acesso em: 19 mar. 2019.

_____. **The Rustonomicon**. 2019. Disponível em: <https://doc.rust-lang.org/stable/nomicon/>. Acesso em: 19 mar. 2019.

_____. **Unsafe Rust**. 2018. Disponível em: <https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html>. Acesso em: 10 mar. 2019.

SANTANNA, Francisco; IERUSALIMSCHY, Roberto; RODRIGUEZ, Noemi. Structured Synchronous Reactive Programming with Ceu, p. 7, 16 mar. 2015. DOI: 10.1145/2724525.2724571.

SCHMIDT, Douglas C. Reactor: An Object Behavioral Pattern forConcurrent Event Demultiplexing and Dispatching, **Proceedings of the1st Annual Conference on the Pattern Languages of Programs**, p. 1–10, ago. 1994. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=DCE9FE741829C28A97DC1FFBE3458A5F?doi=10.1.1.51.6292&rep=rep1&type=pdf>.

SERDE-RS. **Serde**. Abr. 2019. Disponível em: <https://serde.rs/>. Acesso em: 23 abr. 2019.

_____. **syn::Ident**. 2019. Disponível em: <https://docs.serde.rs/syn/struct.Ident.html>. Acesso em: 25 abr. 2019.

SERGIO BENITEZ. **Rocket - Simple, Fast, Type-Safe Web Framework for Rust**. 2018. Disponível em: <https://rocket.rs/>. Acesso em: 23 abr. 2019.

SILVA, Cecília Carneiro e. **Heapless Timer Wheel**. original-date: 2019-02-11T11:04:45Z. 25 mar. 2019. Disponível em: <https://github.com/ceciliacsilva/timer_wheel>. Acesso em: 25 mar. 2019.

_____. **Narc**. original-date: 2018-09-21T18:08:38Z. 11 fev. 2019. Disponível em: <https://github.com/ceciliacsilva/narc>. Acesso em: 18 mar. 2019.

_____. **RTFM-apps - Aplicações embarcadas usando o RTFM.** 2019. Disponível em: <https://github.com/ceciliacsilva/rtfm-app/>. Acesso em: 5 abr. 2019.

_____. **sm-hvac.rs**. original-date: 2018-10-24T18:08:28Z. 5 abr. 2019. Disponível em: <https://github.com/ceciliacsilva/rtfm-app>. Acesso em: 15 abr. 2019.

_____. **State Machine for RTFM.** original-date: 2019-02-05T11:06:17Z. 21 mar. 2019. Disponível em: <https://github.com/ceciliacsilva/sm-rtfm>. Acesso em: 15 abr. 2019.

_____. **STM32L032-rs**. original-date: 2018-10-29T22:20:45Z. 7 dez. 2018. Disponível em: <https://github.com/ceciliacsilva/stm32l052>. Acesso em: 18 mar. 2019.

SONDERGAARD, Harald; SESTOFT, Peter. **P. Acta Informatica**. [S.l.: s.n.], 1990. v. 27. 505 p. Disponível em: <https://doi.org/10.1007/BF00277387>. Acesso em: 14 mar. 2019.

SPRUNT, Brinkley; SHA, Lui; LEHOCZKY, John. **Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System:** Fort Belvoir, VA, 1 abr. 1989. DOI: 10.21236/ADA211344. Disponível em: <http://www.dtic.mil/docs/citations/ADA211344>. Acesso em: 11 mar. 2019.

STMICROELECTRONICS. RM0376 - Reference manual. Ultra-low-power STM32L0x2, p. 1001, dez. 2017.

_____. **STM32L0 - ARM Cortex-M0+ ultra-low-power MCUs - STMicroelectronics**. 2019. Disponível em: <https://www.st.com/en/microcontrollers-microprocessors/stm32l0-series.html?querycriteria=productId=SS1817>. Acesso em: 14 mar. 2019.

STROM, Robert E.; YEMINI, Shaula. Typestate: A programming Language Concept for Enhancing Software Reability. **IEEE Transactions on Software Engineering**, SE-12, p. 157–171, No1 jan. 1986. DOI: 10.1109/TSE.1986.6312929.

TOLNAY, David. **quote - Rust**. 2019. Disponível em: <https://docs.rs/quote/0.6.12/quote/>. Acesso em: 23 abr. 2019.

_____. **syn - Rust**. 2019. Disponível em: <https://docs.rs/syn/0.15.32/syn/>. Acesso em: 23 abr. 2019.

TONY HOARE. **Null References: The Billion Dollar Mistake**. InfoQ. 25 set. 2009. Disponível em: <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>. Acesso em: 10 mar. 2019.

VARIOUS CONTRIBUTORS. **Empowering everyone to build reliable and efficient software.: rust-lang/rust.** [S.l.]: The Rust Programming Language, 21 mai. 2019. original-date: 2010-06-16T20:39:03Z. Disponível em: <https://github.com/rust-lang/rust>. Acesso em: 21 mai. 2019.

_____. **Rust/Unofficial - Builder Pattern**. original-date: 2015-10-15T02:10:14Z. 10 dez. 2018. Disponível em: <https://github.com/rust-unofficial/patterns>. Acesso em: 9 jun. 2019.

VARIOUS CONTRIBUTORS. **Consider adding load/store atomics for targets that don't support compare and swap.** GitHub. 2017. Disponível em: <https://github.com/rust-lang/rust/issues/45085>. Acesso em: 14 mai. 2019.

_____. **Define a Rust ABI**. GitHub. 2015. Disponível em: <https://github.com/rust-lang/rfcs/issues/600>. Acesso em: 13 mai. 2019.

_____. **Procedural Macros**. 1566-proc-macros. original-date: 2014-03-07T21:29:00Z. 15 fev. 2016. Disponível em: <https://github.com/rust-lang/rfcs>. Acesso em: 22 abr. 2019.

_____. **Procedural macros and "hygiene 2.0"**. GitHub. Out. 2018. Disponível em: <https://github.com/rust-lang/rust/issues/54727>. Acesso em: 23 abr. 2019.

_____. **The Rust Standard Library**. std::cell - Rust. 2019. Disponível em: <https://doc.rust-lang.org/std/cell/>. Acesso em: 14 mar. 2019.

VERHULST, Eric et al. **Formal development of a network-centric RTOS: software engineering for reliable embedded systems**. New York: Springer, 2011. 219 p. ISBN 978-1-4419-9735-7. DOI: 10.1007/978-1-4419-9736-4.

WALKER, David. Substructural Type Systems. In: BENJAMIN C. PIERCE (Ed.). **Advanced Topics in Types and Programming Languages**. Cambridge, Massachusetts: MIT Press, 2005. p. 3–43.

WANG, Feng et al. KRust: A Formal Executable Semantics of Rust. **arXiv:1804.10806 [cs]**, 28 abr. 2018. DOI: 10.1109/TASE.2018.00014. arXiv: 1804.10806. Disponível em: <http://arxiv.org/abs/1804.10806>. Acesso em: 10 mar. 2019.

WOLF, Marilyn. **Computers as components: principles of embedded computing system design**. Third edition. Waltham, MA: Elsevier/Morgan Kaufmann, 2012. 500 p. ISBN 978-0-12-388436-7.

_____. Embedded Software in Crisis. **Computer**, v. 49, n. 1, p. 88–90, jan. 2016. ISSN 0018-9162. DOI: 10.1109/MC.2016.18. Disponível em: <http://ieeexplore.ieee.org/document/7383178/>. Acesso em: 7 mar. 2019.

WRIGSTAD, Tobias; CLARKE, Dave. Is the World Ready for Ownership Types? Is Ownership Types Ready for the World? **IWACO: International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming**, p. 4, 2011.