UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Modelagem Formal do Sistema Operacional de Tempo Real FreeRTOS Utilizando o Método B

Stephenson de Sousa Lima Galvão

Orientador: Prof. Dr. David Boris Paul Déharbe

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Sistemas e Computação da UFRN (área de concentração: Ciência da Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências da Computação.

Número de ordem PPgSC: M000
Natal, RN, agosto de 2010

Divisão de Serviços Técnicos

Catalogação da publicação na fonte. UFRN / Biblioteca Central Zila Mamede

Pereira, Fulano dos Anzóis.

Sobre a Preparação de Propostas de Tema, Dissertações e Teses no Programa de Pós-Graduação em Engenharia Elétrica da UFRN / Fulano dos Anzóis Pereira - Natal, RN, 2006

23 p.

Orientador: Sicrano Matosinho de Melo

Co-orientador: Beltrano Catandura do Amaral

Tese (doutorado) - Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica.

1. Redação técnica - Tese. 2. LaTeX- Tese. I. Melo, Sicrano Matosinho de. II. Amaral, Beltrano Catandura do. III. Título.

RN/UF/BCZM CDU 004.932(043.2)

Modelagem Formal do Sistema Operacional de Tempo Real FreeRTOS Utilizando o Método B

Stephenson de Sousa Lima Galvão

Dissertação de Mestrado aprovada em 16 de agosto de 2010 pela banca examinadora composta pelos seguintes membros:

Prof. Dr. David Boris Paul Déharbe (orientador) ..... DIMAP/UFRN

Prof. Dr. Marcel Vinícius Medeiros Oliveira ..... DIMAP/UFRN

Profa. Dra. Aline Maira Santos Andrade ..... UFBA

Dedico esse trabalho aos meus pais,
irmãos e namorada (futura esposa).

Agradecimentos

Agradeço a Deus por:Ter me dado o dom da vida; Ter me dado pais presente e incentivadores; Ter me dado irmão companheiros, Stanley e Stênio; Ter colocado na minha vida uma companheira ideal, Luanne; Ter me apresentado amigos verdadeiros; Dar-me a oportunidade de trabalhar com verdadeiros mestres pacientes, David; Dar-me a oportunidade de trabalho em um ambiente amigável, IFPI; e Ter me dado forças para concluir esse trabalho.

Resumo

Este trabalho apresenta uma contribuição para o esforço internacional do *Verified Software Repository*, o qual tem como alguns de seus objetivos acelerar o desenvolvimento das tecnologias de verificação de software e colecionar especificações de diversas aplicações como *smart card*, bibliotecas padrões de desenvolvimento e sistema operacionais embarcados. Para essa contribuição, será adotada a sugestão do pesquisador britânico Woodcock de especificar formalmente o sistema operacional de tempo real FreeRTOS, o qual tem como características a sua simplicidade, portabilidade e popularidade. Com essa especificação será possível verificar o correto funcionamento desse sistema e, consequentemente, garantir uma maior confiabilidade para as aplicações desenvolvidas através dele, cooperando, com isso, também para o desafio da construção de sistemas disponíveis, seguros, escalonáveis e ubíquos, proposto pela Sociedade Brasileira de Computação. O formalismo utilizado nessa especificação é o método B, que, devido as suas semelhanças com as linguagens de programação imperativas e suas técnicas de modularização e refinamento, tornou-se viável para a construção da mesma. O modelo desenvolvido teve como principal foco as funcionalidades e abstrações de hardware do sistema e o seu desenvolvimento foi feito de forma incremental. Para provar a viabilidade desse trabalho, uma especificação abstrata foi inicialmente desenvolvida, na qual algumas propriedades do sistema puderam ser tratadas. Ao final, além de validar os requisitos desse sistema, esse trabalho servirá também como uma documentação formal do mesmo, a qual poderá ser usada como entrada para a geração de testes, no nível de código, para o sistema especificado. Em suma, a especificação do FreeRTOS auxiliará na resolução dos desafios anteriormente citados, servindo desse modo como uma contribuição para a sociedade de computação, em especial a sociedade de métodos formais.

Palavras-chave: Especificaçõa,FreeRTOS, Método B, Métodos Formais.

Abstract

This work present a contribuition to the international effort of the Verified Software, that as a objective accelerate the development of verification software technologies and collect various applications especification such as smart cards, library development standards and operating systems embeded. For this contribution, will be adopted the Woodcock, British researcher, suggestion of make a formal especification of the real time operating system FreeRTOS, a simplicity, portability and popularity operation system. This specification will veryfite the correct operation of this system and thus ensure greater reliability for the applications developed through it. This specification, also, will contribui to the challenge of building systems available, secure, scalable, and ubiquitous, proposed by the Company Brazilian Computer. The formalism used for this specification is the B method, wich look like with the imperative programming languages. Further than this, the b method techniques of modularization and refinement became feasible to build specification through it. The model developed had main focus the features and abstractions of system hardware. The specification was done incrementally. To prove the feasibility of this work, an abstract specification was initially developed, in which some properties of the system could be treated. In the end, fhurther than validate the requirements of this system, this work will also serve as a formal documentation of this system, which can be used as input for the tests generation level code to the system. In short, the specification of FreeRTOS assist in resolving the challenges cited above, thereby serving as a contribution to society of formal methods.

Keywords: Especifitacion, FreeRTOS, B Method, Formal Method.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Ciência da Computação é uma área relativamente jovem, mas com grande impacto na sociedade atual. Através dela, é possível acelerar-se anos de desenvolvimento e pesquisas das demais áreas da ciência, como ocorreu com o mapeamento do genoma humano. Devido a isso, em [**?**] ela chegou a ser citada como um dos pilares da ciência, considerada também como um fator crucial para a economia, tecnologia e desenvolvimento de um pais.

Ciente dessa importância, pesquisadores de todo mundo tem buscado reunir esforços em torno de objetivos comuns, definindo diretrizes a serem seguidas pela computação ao longo dos próximos anos. Essas diretrizes são estabelecidas visando suprir as necessidades mais relevantes da sociedade atual, as quais, em geral, são obstáculos ainda não vencidos pela computação, e, por isso, denominados como "Grandes Desafios da Computação"

Seguindo esse contexto, a Sociedade Brasileira de Computação (SBC) listou, em [**?**], alguns desses principais desafios, dentre os quais está o desafio de *desenvolvimento tecnológico de qualidade: sistemas disponíveis, corretos, seguros, escaláveis, persistentes e ubíquos*. Esse desafio visa a pesquisa de métodos, técnicas, modelos, dispositivos e padrões de projeto capazes de auxiliar os projetistas e desenvolvedores de grandes sistemas de software e hardware a criarem produtos mais previsíveis, escalonáveis e seguros.

A inspiração para o desenvolvimento tecnológico de qualidade veio principalmente da crescente utilização de sistemas de computação no cotidiano em eletrodomésticos, celulares, carros, metrôs, aviões e salas de cirurgia. Alguns desses são sistemas relativamente simples e não causam grandes problemas em caso de falhas, outros são críticos e uma falha pode levar à consequências catastróficas. Para esses últimos, o desenvolvimento de sistemas fidedignos é de extrema importância.

Entre as abordagens de construção de sistemas fidedignos, os métodos formais tem sido um caso de sucesso. Métodos formais são técnicas matemáticas usadas para desenvolver sistemas de *software* e *hardware* com rigor matemático. Eles permitem ao usuário analisar e verificar seus modelos em todas as partes do ciclo de vida de desenvolvimento: requisitos, especificação, modelagem, implementação, teste e manutenção. Atualmente, esses métodos têm sido utilizados,

de forma auspiciosa, pela indústria para a construção de aplicações pequenas, mas significantes [**?**]. Pontualmente, sistemas de grande porte foram analisados usando técnicas formais no projeto astrée[1].

Entretanto, a comunidade de métodos formais também possui seus desafios, entre eles está o projeto do "Software Verificado"[**?**]. O Software Verificado é um conjunto de teorias, ferramentas e experiências usadas para verificação e validação de software. Ele tem como objetivo aplicar técnicas existentes a exemplos significativos para identificar limites, comparar as capacidades e acelerar o amadurecimento das tecnologias de verificação de software. Para isso, foi criado um repositório onde são armazenadas a utilização de diferentes abordagens de verificação aplicadas estudos de casos sugeridos pelo projeto, dos quais encontra-se o sistema operacional de tempo real FreeRTOS[**?**][**?**].

O FreeRTOS [**?**][**?**] é um sistema operacional de tempo real popular, simples e portável. Atualmente, o seu repositório[**?**] possui uma taxa maior que 6.000 downloads por mês; o seu núcleo é acessível, aberto e pequeno. São aproximadamente 2.200 linhas de código, com funções comuns a maioria dos sistemas operacionais. Ele oficialmente suporta 17 arquiteturas de microcontroladores diferentes. Devido a essas características, a especificação desse sistema seria uma contribuição para a comunidade de métodos formais.

Entre as possíveis abordagens formais para a especificação do FreeRTOS está o método B [**?**][**?**]. O método B é uma abordagem de especificação, validação e construção de sistemas. Nele o sistema é inicialmente especificado em uma linguagem matemática parecida com uma linguagem de programação e, em seguida, são geradas várias obrigações de prova para comprovar a coerência dessa especificação. Além disso, para facilitar o desenvolvimento de sistemas, ele dispõe dos mecanismos de modularização e refinamento, sendo esse último capaz de refinar uma especificação até um nível algoritmo, passível de transformação para as linguagens de programação imperativas. Por último, o método B é apoiado por um pacote de ferramentas que suportam todos os seus ciclos de desenvolvimento, o que tornou essa abordagem a escolhida para a especificação do FreeRTOS nesse trabalho.

Em suma, a especificação formal do FreeRTOS através do método B, além ser um esforço para a resolução dos desafios apresentados, é uma grande contribuição para a computação, principalmente a comunidade de métodos formais. Tal especificação poderá ainda agregar valor ao sistema, servindo como documentação e entrada para técnicas de teste de sistemas construídos utilizando o Sistema Operacional em questão [**?**]. Além disso, devido a utilização do método B em um caso de uso real, esse trabalho pode apresentar problemas de limitações do método B, para o aperfeiçoamento do mesmo.

[1] http://www.astree.ens.fr/

1.1 Objetivo

O objetivo desse trabalho é, através da especificação do FreeRTOS em B, contribuir para os desafios do software verificado e da construção de software fidedigno. A especificação desenvolvida modelará funcionalmente, de acordo com a documentação, algumas das principais funcionalidades que compõem o FreeRTOS, sendo capaz de analisar propriedades estáticas e levantar questionamentos sobre o funcionamento das mesma. Além disso, com a continuação desse trabalho será possível desenvolver um sistema operacional de tempo real, fundamentado na documentação do FreeRTOS, criado por uma especificação em B.

1.2 Metodologia

A metodologia utilizada no desenvolvimento desse trabalho pode ser sucintamente resumida pelos itens abaixo, os quais serão comentados nos parágrafos seguintes:

- Estudo do método B;
- Estudo da documentação e código fonte do FreeRTOS;
- Elaboração de um plano para a modelagem funcional do FreeRTOS;
- Modelagem funcional do FreeRTOS considerando apenas as entidades Tarefa e Fila de Mensagens;
- Refinamento da modelagem desenvolvida para adicionar a característica de prioridade de uma tarefa e tamanho da fila de mensagens;
- Amadurecimento da modelagem desenvolvida;
- Acréscimo das entidades mutex e semáforo à especificação; e
- Refimento da modelagem;

Pelo fato do método B ser o formalismo escolhido para o desenvolvimento da modelagem proposta, o primeiro passo desse trabalho foi a pesquisa sobre essa abordagem. Durante essa pesquisa, vários estudos de caso foram desenvolvidos, entre eles uma especificação do microcontrolador 8051 (melhor detalhada em [**?**]). Além disso, com o conhecimento adquirido, foi possível colaborar com os seguintes trabalhos [**?**] e [**?**].

O estudo sobre o FreeRTOS foi divido em duas partes principais: a pesquisa na documentação do sistema e a análise do código fonte do mesmo. Essas partes serão detalhadas nos parágrafos seguintes.

No estudo da documentação do sistema, os principais requisitos, funcionalidades e características do FreeRTOS puderam ser identificados e classificados de acordo com sua importância. Esse trabalho serviu como base para a análise do código fonte e para a elaboração do plano de desenvolvimento de uma modelagem funcional do sistema.

Durante a análise do código fonte, foi possível identificar a arquitetura do sistema e como suas principais abstrações de hardware e funcionalidades são implementadas. Nessa análise, percebeu-se que muita vezes o sistema possui detalhes

em níveis de assemblagem, o que segundo Dantas, em [**?**], também podem ser tratados utilizando o método B.

Através da análise do FreeRTOS descobriu-se que, apesar dele ser um sistema simples e pequeno, a sua modelagem é um trabalho que levará muito tempo e esforço. Além disso, limitações existentes no método B impedem a sua completa especificação. Assim, para o melhor aproveitamento do modelo desenvolvido, foi elaborado um plano para identificar quais elementos e funcionalidades que devem fazer parte dessa especificação.

No plano de modelagem, determinou-se que a especificação do sistema será feita em etapas. Desse modo, entidades do FreeRTOS escolhidas e suas características foram acrescentadas de forma incremental, amadurecendo a especificação. Devido a isso, na modelagem inicial, foram especificadas apenas as entidades tarefa e fila de mensagens, sendo as demais entidades escolhidas (semáforo e mutex) tratadas nas etapas seguintes da especificação.

Com a modelagem inicial do sistema, foi iniciada a especificação das funcionalidades eleitas do FreeRTOS. Assim, utilizando-se do mecanismo composicional do método B, definiu-se a arquitetura da especificação. Nela, o sistema foi divido em sete módulos principais, organizados de acordo com a figura 1.1. Nesta figura os módulos mais abaixo fornecem operações e implementam elementos para os módulos superiores realizarem suas funções, sendo o módulo *FreeRTOS* responsável por especificar as funcionalidades do sistema.

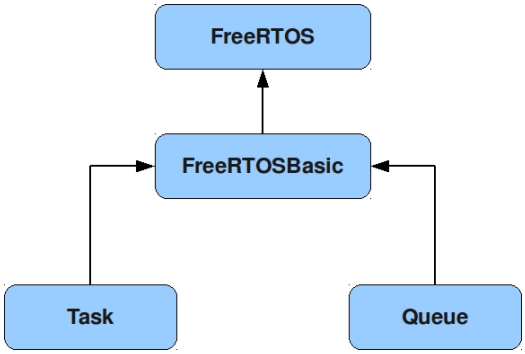


Figura 1.1: Esboço da arquitetura da especificação.

Entretanto, por tratar-se de um modelo funcional, a modelagem inicial abstraiu muitas características importantes do FreeRTOS, entre elas a característica de prioridade de uma tarefa e tamanho de uma fila de mensagens. A adição dessas características ao modelo foi feita em seguida, utilizando o mecanismo de refinamento do método B, no qual uma especificação abstrata é refinada para um modelo mais concreto. Nessa etapa foram criados dois novos módulos, um para a prioridade de uma tarefa e outro para o tamanho de uma fila de mensagens.

Ao total, nessa especificação desenvolvida pela primeira etapa de refinamento, foram criadas 1.974 linhas, as quais geraram 538 obrigações de prova (seções 4.3). Entre essas obrigações de prova, somente 49 necessitaram da interação humana

para sua resolução, sendo as demais provadas automaticamente pela ferramenta adotada pelo projeto[**?**].

Dando continuidade na metodologia do trabalho, as próximas etapas são: amadurecimento da modelagem desenvolvida; acréscimo das entidades mutex e semáforo à especificação; refinamento da modelagem; e amadurecimento e encerramento da especificação.

Após o refinamento das etapas anteriores, a especificação atingiu um nível razoavelmente maduro, mas devido a dificuldades de obrigações de prova a especificação foi criada com abstrações sobre as características dos elementos especificados. Para uma modelagem mais completa desses elementos, foi necessário, nessa etapa, reformular toda a especificação deixando-a mais madura, mas seguindo a mesma arquitetura.

Com o amadurecimento das entidades tarefas e fila de mensagens, iniciou-se a especificação dos demais elementos na etapa de acréscimo das entidades mutex e semáforo à especificação. Nessa etapa, devido à proximidade dessas entidades com a fila de mensagens, os novos elementos foram adicionados ao módulo dessa entidade, necessitando reformulá-lo.

Terminada a adição das entidades escolhidas para a especificação foi feito o refinamento dos novos elementos e, em seguida, a adição de suas características, tornando a modelagem mais concreta e madura. Nessa etapa, a especificação encontrava-se bastante sólida evitando grandes reformulações da especificação para abrigar os novos requisitos.

Ao final, criou-se uma especificação, concreta do FreeRTOS, na qual funcionalidades do sistema puderam ser testadas. Além disso, a especificação aqui desenvolvida servirá como material de apoio ao FreeRTOS, gerando uma documentação do sistema no ponto de vista formal e poderá ser incrementada para o nível algorítmico, criando um sistema operacional de tempo real especificado em B.

Continuando essa introdução, a dissertação está organizada da seguinte forma. No capítulo seguinte, encontra-se alguns trabalhos importantes que serviram como base para este. Em seguida, uma introdução sobre o FreeRTOS e o Método B são mostradas nos capítulos 3 e 4, respectivamente. Após isso, a especificação iniciamente desenvolvida do FreeRTOS é demonstrada no capítulo 6 e a continuação da mesma é expandinda no capítulo 7. Por fim, tem-se as lições aprendidas e as conclusões no capítulo 8.

Capítulo 2

Trabalhos Relacionados

Vários foram os trabalhos que tiveram como objetivo especificar e verificar sistemas operacionais (SO), mais especificamente micronúcleo (ou *microkernel*) de SO, parte do sistema operacional que implementa suas funcionalidades, classificação na qual encontra-se o FreeRTOS também. Desses trabalhos, pode-se tirar proveito em técnicas e estratégias que foram úteis nessa dissertação. Assim, para o melhor entendimento, os trabalhos relacionados, serão organizados inicialmente por ordem cronológica (trabalhos anteriores e trabalhos recentes) e, em seguida, será mostrado na seção 2.3, alguns esforços que são relevantes.

2.1 Trabalhos Anteriores

Os esforços para especificação e verificação de sistemas operacionais não são recentes, entre os anos 70 e 80 surgiram três trabalhos pioneiros relacionados a esse tema. São eles: [**?**], [**?**] e [**?**].

Em [**?**], Popek documenta a especificação e validação do UCLA Secure Data Unix, sistema operacional utilizado pelas aplicações do computador DEC PDP-11/45. Nesse trabalho, os esforços de verificação foram focados no núcleo do sistema, que provê serviços semelhantes aos atuais *microkernels*, como tarefas e paginação. Além disso, o projeto foi um dos primeiros a separar política do sistema operacional do mecanismo do núcleo do sistema, o que facilitou a verificação do sistema.

No trabalho de Popek, alguns pontos importantes foram utilizados nessa dissertação. Inicialmente, ele dividiu a sua especificação em camadas iguais a figura 2.1, na qual a única diferença entre elas está no nível de abstração. Assim, cada camada provê todos os detalhes do sistema em níveis de abstrações diferente. Por exemplo, o nível de código possui todas as variáveis usadas nas chamadas de funções do sistema, mas usa menos estruturas abstratas que as camadas superiores da especificação, como listas e conjuntos.

A prova da coerência entre as camadas do UCLA foi feita como demonstra a figura 2.2, através do mapeamento de estados. Nesta, cada camada possui uma máquina de estado, um conjunto de estados e suas transições, sendo que cada máquina deve simular as demais. Com isso, deve existir uma função que mapeie

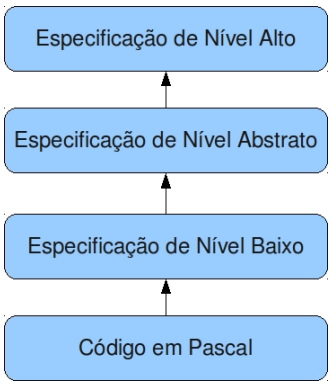


Figura 2.1: Camadas da especificação do UCLA.

os estados do nível concreto com os estados do nível abstrato. Apesar de não ser classificada como esse termo por seus autores, a técnica utilizada por Popek foi o refinamento formal [**?**].

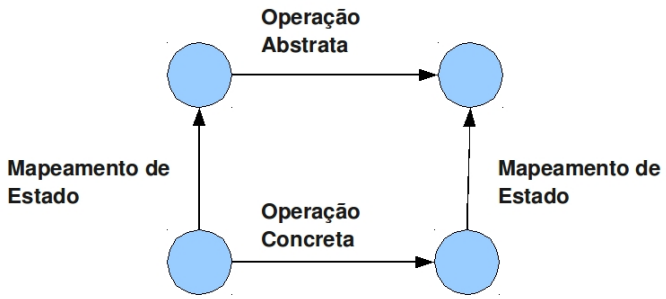


Figura 2.2: Mapeamento de estado entre o níveis de abstração.

No [**?**], o sistema PSOS foi projetado. Esse foi um trabalho maior que o UCLA, pois, diferente de focar somente no núcleo, as regras particulares do sistema operacional também foram consideradas. A grande contribuição desse trabalho foi a diferente divisão de camadas do projeto do sistema. As camadas mais baixas oferecem funcionalidades para as camadas superiores. Ao total, foram desenvolvidas 17 camadas, como demonstra a figura 2.1, sendo as 6 inferiores projetadas para serem implementadas no hardware.

Para projetar o PSOS, inicialmente foi criada uma linguagem de especificação e sentenças chamada SPECIAL. Essa linguagem foi utilizada para especificar formalmente os módulos do sistema, que formavam as suas camadas. SPECIAL também permitia o mapeamento de funções e implementações abstratas, relacionando módulos entre níveis.

Vale ressaltar que, apesar das várias tentativas, os trabalhos anteriores foram dificultados pelo número de mecanismos e ferramentas apropriadas disponíveis na época. Assim, enquanto o projeto era formalizado, a completa verificação de

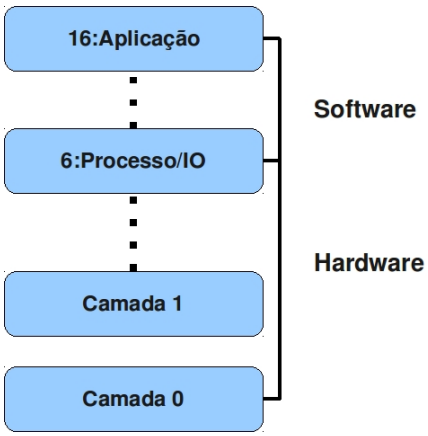


Figura 2.3: Camadas da especificação do PSOS.

prova não era desenvolvida. De acordo com Klein, em [**?**], apenas vinte por cento do UCLA foi provado e projeto PSOS não foi provado.

Diferente dos trabalhos anteriores, [**?**] se destaca por ser o primeiro trabalho a especificar e provar formalmente o *microkernel* de um Sistema Operacional. O sistema especificado foi o KIT, um micronúcleo simples e com menos abstrações de hardware que os *microkernels* modernos. Em seu trabalho, Bevier conseguiu a taxa de cem por cento de verificação e prova da especificação.

2.2 Trabalhos Recentes

Entre os trabalhos recentes de especificação de sistemas operacionais tem-se os projetos VFiasco, o Coyotos, o Verisoft e o Verificação do L4.

VFiasco[**?**] é um projeto de especificação formal do Fiasco, uma implementação mais simples do *microkernel* L4. Um trabalho importante desse projeto foi listar alguns dos problemas cruciais da verificação dos modernos *microkernels*. Entre esses problemas, está a semântica formal da linguagem de implementação e a complexidade da memória de execução do núcleo, pois é ele quem implementa os mecanismos de gerenciamento de memória como paginação.

Além disso, outra contribuição do projeto VFiasco foi o desenvolvimento de uma memória virtual de comportamento bem definido, sobre a qual o sistema será executado, pois as memórias de hardware não possuem comportamento especificada, o que dificuta a verificação do sistema. Assim, tal solução poderá ser aproveitada na especificação formal dos demais sistemas operacionais, que também são executados em memórias de comportamento não previsível.

No projeto Coyotos [**?**] foi desenvolvida uma implementação formal para o sucessor do *microkernel* ERO, o Coyoto. Uma das características centrais desse projeto foi a criação de uma nova linguagem de programação para a implementação do núcleo a BitC. Essa linguagem teve como objetivos ser segura, clara e

passível de verificação e foi desenvolvidas agrupando características das pesquisas existentes.

Em seguida, tem-se o projeto Verisoft que é um esforço para verificação de toda uma pilha de sistemas, a qual varia de hardwares até aplicações, passando por compiladores e *microkernels*. Nesse projeto, encontra-se a especificação formal do *microkernel* simplificado VAMOS.

Por último e mais interessante está o projeto de especificação verificação do L4, um *microkernel* simples e pequeno. Nesse projeto, encontra-se o trabalho de Kolanski [**?**], que bastante se parece com essa dissertação. Nesse trabalho, é feita a especificação da Interface de Programação de Aplicação (API - Application Programming Interface) do *microkernel* no ponto de vista do sistema, onde as chamadas as funções do núcleo são formadas por alterações nos estados do núcleo que gerencia as abstrações por ele criadas.

Os pontos comuns entre [**?**] e a especificação do FreeRTOS começa com a semelhança dos sistemas, os quais são sistemas usados pela indústria, e continua no objetivo, que é especificar formalmente um *microkernel*, classificação na qual está também o FreeRTOS, no ponto de vista interno do sistema. Além disso, Kolanski também escolheu o método B como formalismo para a especificação e organizou o seu trabalho em módulos responsáveis pelas abstrações de hardware do sistema, os quais estão relacionados pelo mecanismo de inclusão de B, como demonstra a figura 2.2.

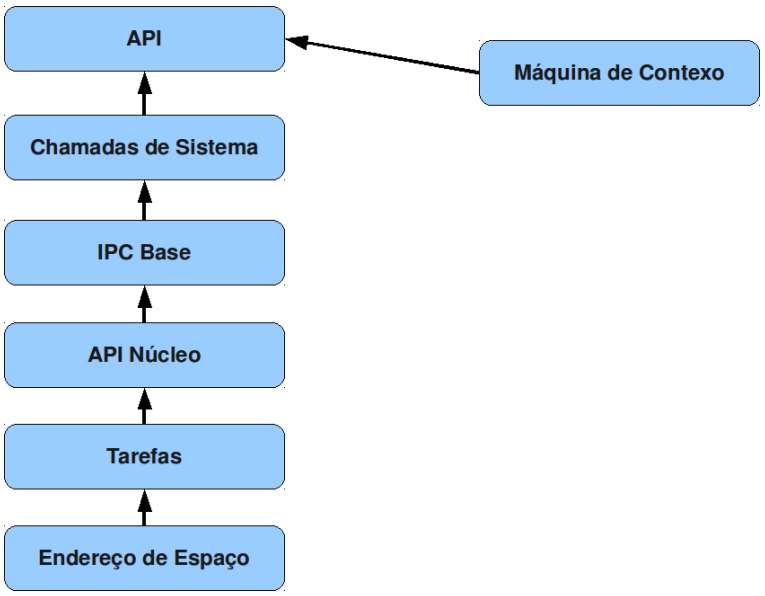


Figura 2.4: Módulos da especificação do L4.

A diferença do trabalho de Kolanski e a especificação do FreeRTOS está na estratégias de especificação. No seu trabalho, Kolanski especificou de forma abstrata a API do microkernel L4, enquanto que na especificação do FreeRTOS, além

de especificação abstrata, níveis de refinamentos foram criados para dar um nível mais concreto ao modelo criado.

2.3 Outros trabalhos

Um trabalho que está mais preocupado com concorrência entre processos de uma aplicação, mas que deve ser comentado é [**?**]. Nesse trabalho, a validação de uma aplicação concorrente é feita através da transformação da linguagem intermediária do compilador LLVM para uma modelagem de baixo nível em CSP. Esse modelo é dividido em três partes: Aplicação, onde o comportamento dos processos do sistema são descritos; Domínio, onde os aspectos comuns do domínio da aplicação são especificados; e Plataforma, onde os detalhes da plataforma utilizada são formalizados. Um fator positivo dessa divisão é que essa especificação pode disponibilizar diversas visões do sistema e as partes de domínio e plataforma podem ser parametrizáveis, podendo assim ser reutilizadas em várias aplicações.

Após a criação do modelo CSP, este é analisada através das ferramentas de verificação para CSP, como FDR2 e ProB, que verificam situações de *deadlock* e condições de corrida do sistema. Além disso, com a geração do modelo CSP, as atividades implementadas no sistema são abstraídas para um ponto de vista mais prático e abstrato, permitindo assim uma melhor análise do sistema.

Por fim, tem [**?**], que é um guia para a especificação de sistemas operacionais. Nesse trabalho, Craig começa afirmando que a parte mais importante de um sistema é o sistema operacional, que gerencia os recursos usados pela aplicação. Após isso, ele certifica que os métodos formais há tempos estão relacionados com os sistemas operacionais e que, na maioria das vezes, foram utilizados para especificar as operações de fila dos sistemas. De posse dessas afirmações, Craig defende a especificação formal como uma prática de suma importância a ser realizada antes da codificação, pois através dela o sistema pode ser analisado de forma abstrata, como uma entidade matemática. Assim, propriedades importantes do sistema podem ser provadas antes mesmo de sua codificação, o que diminui os riscos do projeto.

Continuando o trabalho, Craig demonstra a divisão em camadas de um sistema operacional convencional, sendo essas: a camada das primitivas de *hardware*, localizada acima do *hardware*; a camada responsável pelo gerenciamento de disco e interrupções de *hardware* (Relógio do sistema); a camada de gerenciamentos de arquivos e controle de interfaces; e a camada de chamadas do sistema utilizada pelas aplicações. Essa divisão proporciona uma modelagem incremental de um SO, na qual os elementos mais abstratos e importantes são tratados inicialmente.

Finalmente, através dos trabalhos aqui relacionados, percebe-se que, devido à importância do correto funcionamento dos sistemas operacionais, vários traba-

lhos de especificação e verificação destes tem sido desenvolvidos. Nesses trabalhos, várias foram as estratégias, mas uma observação comum a todas elas foi a simplificação e modularização dos sistemas desenvolvidos, criando camadas e focando em partes específicas dos sistemas. Essas estratégias foram o marco inicial para a especificação do FreeRTOS.

Capítulo 3

FreeRTOS

O FreeRTOS é um *microkernel* sistema de tempo real enxuto, simples e de fácil uso. O seu código fonte, em *C* com partes em *assembly*, é aberto e possui um pouco mais de 2.200 linhas de código, que são essencialmente distribuídas em quatro arquivos: *task.c*, *queue.c*, *croutine.c* e *list.c*. Outra característica marcante desse sistema está na sua portabilidade, sendo o mesmo oficialmente portável para 17 arquiteturas mono-processadores diferentes, entre elas a PIC, ARM e Zilog Z80, as quais são amplamente difundidas em produtos comerciais através de sistemas computacionais embutidos [**?**].

Como a maioria dos *microkernel* atuais, o FreeRTOS provê, para os seus usuários, acesso facilitado aos recursos de *hardware*, agilizando com isso o desenvolvimento de sistemas de tempo real. Desse modo, ele funciona como na figura 3.1, uma camada de abstração, localizada entre a aplicação e o hardware, que tem o papel de esconder dos desenvolvedores de aplicações detalhes do hardware, no qual as aplicações serão utilizadas[**?**].

Figura 3.1: Camada abstrata proporcionada pelo FreeRTOS.

Para prover tal abstração, o FreeRTOS possui um conjunto de bibliotecas de tipos e funções que devem ser linkeditadas[1] com o código da aplicação a ser desenvolvida. Juntas, essas bibliotecas fornecem aos desenvolvedores serviços como gerenciamento de tarefa, comunicação e sincronização entre tarefas, gerenciamento de memória e controle dos dispositivos de entrada e saída[**?**].

[1]Processo que liga o código da aplicação ao código das funcionalidades de outras bibliotecas utilizada por ela.

Devido a sua portabilidade e ao fato de ser usado em ambientes com limitações de hardware, o FreeRTOS pode ser pré-configurado antes da sua execução. Essa configuração é feita por uma biblioteca de configuração, que, através de atributos, armazena as definições de configuração do usuário. Com isso, as aplicações desenvolvidas com o FreeRTOS podem ser mais enxutas e moldadas, provendo uma melhor utilização dos recursos de hardware.

Nas seções a seguir, será explicado, em maiores detalhes, os principais serviços providos pelo FreeRTOS, assim como as bibliotecas e funções que os disponibilizam.

3.1 Gerenciamento de Tarefas e Co-Rotinas

Sabe-se que uma aplicação é formada por várias rotinas, responsáveis por realizar as funcionalidades da aplicação. No FreeRTOS, as unidades responsáveis por abrigar essas rotinas são as tarefas e co-rotinas. Nessa seção será demonstrado como esses elementos funcionam, suas diferenças e unidade de gerenciamento. Ao final, as funcionalidades responsáveis pelo controle dessas abstrações serão listadas.

3.1.1 Tarefa

Para entender como funciona o gerenciamento de tarefas do FreeRTOS, é necessário primeiramente entender o conceito de tarefa. Tarefa é uma unidade básica de execução que compõe os sistemas, os quais, para realizar suas atividades, geralmente possuem várias tarefas com diferentes obrigações[**?**]. Para o FreeRTOS, as principais características de uma tarefa são:

Estado: Demonstra a atual situação da tarefa;
Prioridade: Indica a importância da tarefa para o sistema. Uma prioridade varia de zero até uma constante máxima pré configurada pelo projetista;
Pilha de execução: Local onde uma tarefa armazena informações necessárias para a sua execução;
Contexto próprio: Capacidade de uma tarefa armazenar o ambiente de execução quando suas atividades são suspensas; e
Tempo de bloqueio: Tempo que uma tarefa pode permanecer bloqueada a espera de algum evento.

Em um sistema, uma tarefa pode assumir vários estados, que variam de acordo com a sua situação. O FreeRTOS disponibiliza quatro tipos de estados diferentes para uma tarefa, sendo eles:

Em execução: Indica que a tarefa está em execução;
Ponta: Indica que a tarefa está pronta para entrar em execução, mas não está sendo executada;

Bloqueada: Indica que a tarefa está esperando por algum evento para continuar a sua execução; e

Suspensa: Indica que a tarefa foi suspensa pelo gerenciador de tarefas através da chamada de uma funcionalidade usada para controlar as tarefas.

No FreeRTOS, uma tarefa só possui um estado em um determinado instante. Assim, as alterações de estado de uma tarefa funcionam como demonstra o diagrama da figura 3.2. Nela, uma tarefa com o estado em execução pode assumir os estados pronta, bloqueada ou suspensa. Uma tarefa com o estado pronta pode ser suspensa ou entrar em execução, e as tarefas com o estado bloqueada ou suspensa só podem ir para o estado pronta.

Um fator preocupante, observado nas possíveis troca de estados de uma tarefa, ocorre quando uma tarefa bloqueada é suspensa e, logo em seguida, reativada. Assim o tempo de bloqueio de uma tarefa pode ser "enganado" e a tarefa pode retornar antes do tempo indicado para o desbloqueio.

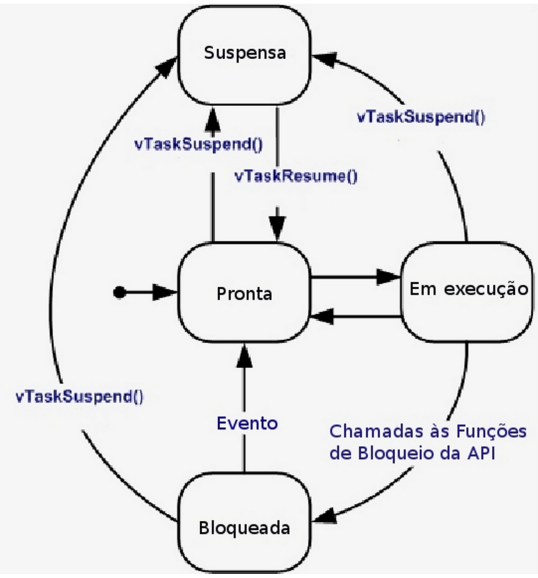


Figura 3.2: Diagrama de estados de uma tarefa no FreeRTOS.

Por fim, vale enfatizar que, por tratar-se de um sistema operacional para arquiteturas mono-processadores, o FreeRTOS não permite que mais de uma tarefa seja executada ao mesmo tempo. Assim, em um determinado instante, apenas uma das tarefas com estado pronta pode assumir o processador e entrar no estado em execução. Com isso, para decidir qual tarefa entrará em execução, o FreeRTOS possui um mecanismo denominado escalonador, o qual será detalhado na seção 3.1.3.

Tarefa Ociosa

No FreeRTOS existe uma tarefa especial, denominada Tarefa Ociosa, que é executada, como o nome sugere, quando o processador encontra-se ocioso, ou

seja, quando nenhuma tarefa estiver em execução. Essa tarefa tem como principal funcionalidade liberar área de memórias que não estão sendo mais utilizadas pelo sistema. Por exemplo, quando uma aplicação cria uma nova tarefa, uma área da memória é reservada a ela. Em seguida, quando essa tarefa é excluída do sistema, a memória destinada a ela continua ocupada, sendo esta liberada somente quando a tarefa ociosa entra em execução. A tarefa ociosa deve possuir prioridade menor que as demais tarefas do sistema e, por isso, ela só é executada quando nenhuma tarefa estiver em execução.

3.1.2 Co-rotinas

Outro conceito importante suportado pelo FreeRTOS é o conceito de Co-rotinas, assim como as tarefas, são unidades de execução independentes que formam uma aplicação. Elas também são formadas por uma prioridade e um estado responsáveis, respectivamente, pela importância da co-rotina no sistema e pela situação da mesma. Para o FreeRTOS, o suporte às co-rotinas é opcional e pré-configurável antes da execução, em tempo de compilação [**?**].

Uma diferença crucial entre co-rotinas e tarefas está no contexto do ambiente de execução. Co-rotinas não possuem contexto de execução próprio. Consequentemente, sua pilha de execução é compartilhada com as demais Co-rotinas do sistema, diferente das tarefas, que possuem uma pilha própria para o armazenamento do seu contexto de execução.

Devido ao fato de co-rotinas compartilharem a mesma pilha de execução, a utilização dessa entidade deve ser feita de forma cuidadosa, pois uma informação armazenada por uma co-rotina pode ser alterada por outra.

Os estados possíveis para uma co-rotina são:

Em execução: Indica que uma co-rotina está em execução;
Pronta: Indica que uma co-rotina está pronta para ser executada, mas não está em execução; e
Bloqueada: Indica que a co-rotina está bloqueada esperando por algum evento para continuar a sua execução.

Como as tarefas, co-rotinas possuem somente um estado em um determinado instante. Assim, as transições entre os estados de uma co-rotina ocorrem como demonstra a figura 3.3. Nessa, uma co-rotina em execução pode ir tanto para o estado pronta como para o estado bloqueada. Uma co-rotina de estado bloqueada só pode ir para o estado pronta e uma co-rotina de estado pronta só pode ir para o estado em execução.

Assim como nas tarefas, a decisão de qual co-rotina irá entrar em execução é feita pelo escalonador, através da chamada a uma funcionalidade específica, esse processo será melhor explicado na seção a seguir.

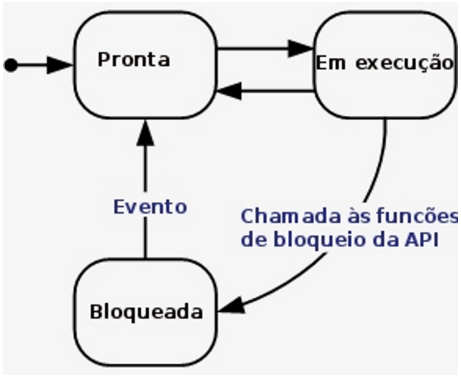


Figura 3.3: Grafo de estados de uma co-rotina.

3.1.3 Escalonador de Tarefas

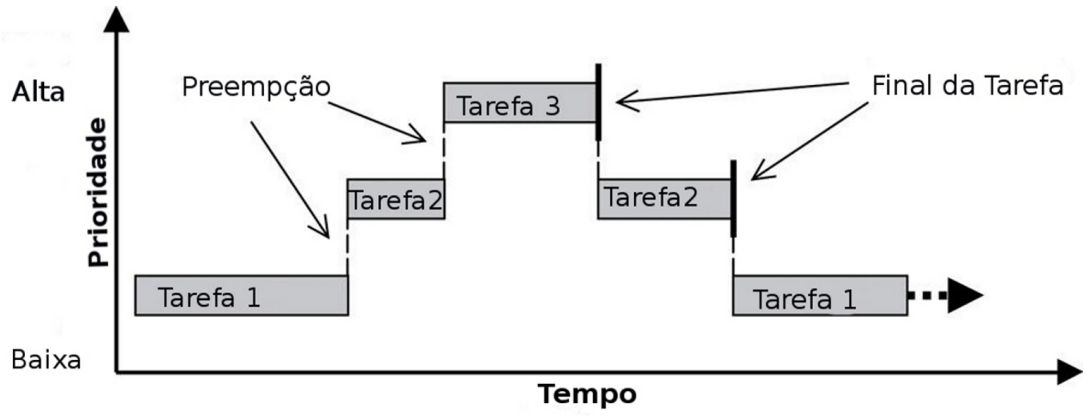
O escalonador é a parte mais importante de um sistema. Ele é responsável por escolher qual unidade de execução[2] irá entrar execução. Além disso, é ele que faz a troca entre a unidade em execução e a nova unidade que irá entrar em execução. No FreeRTOS, o escalonador pode funcionar de três modos diferentes, configuráveis em tempo de compilação:

- **Preemptivo**: Quando o escalonador interrompe a unidade em execução, alterando assim o seu estado, e ocupa o processador com outra unidade;
- **Cooperativo**: Quando o escalonador não tem permissão de interromper a unidade em execução. Assim, a interrupção da execução da unidade em processamento e a chamada ao escalonador para decidir quem irá entrar em execução devem ser implementadas pelo projetista; e
- **Híbrido**: Quando o escalonador pode comportar-se tanto como preemptivo como cooperativo.

Para as tarefas, o escalonador funciona de forma preemptiva, sendo que a decisão de qual tarefa deve entrar em execução é baseada na prioridade e adota a seguinte política: a tarefa em execução deve ter prioridade maior ou igual as tarefas de maior prioridade de estado pronta. Assim, sempre que uma tarefa, com prioridade maior que a tarefa em execução, entrar no estado pronta, ela deve imediatamente entrar em execução. Um exemplo claro da política preemptiva pode ser visto na figura 3.4, onde três tarefas, em ordem crescente de prioridade, disputam a execução do processador.

Um fato a adicionar sobre a política de funcionamento do escalonador é que, quando duas ou mais tarefas de estado pronta tiverem prioridades iguais e maiores que as demais tarefas do mesmo estado, o tempo de execução será divido entre essas tarefas. Assim, ao possuir duas tarefas de prioridades máximas e estado pronta, essas tarefas irão alternadamente ser executadas pelo processador.

[2]Aqui o termo "unidade de execução", as vezes citado apenas como unidade, refere-se a todas as tarefas(seção 3.1.1) e co-rotinas (seção 3.1.2) do sistema



1. Tarefa 1 entra no estado pronta, como não há nenhuma tarefa em execução esta assume controle do processador entrando em execução.
2. Tarefa 2 entra no estado pronta, como esta tem prioridade maior do que a tarefa 1 ela entra em execução passando a tarefa 1 para o estado pronta.
3. Tarefa 3 entra no estado pronta, como está tem prioridade maior do que a tarefa 2 ela entra em execução passando a tarefa 2 para o estado pronta.
4. Tarefa 3 encerra a sua execução, sendo a tarefa 2 escolhida para entrar em execução por ser a tarefa de maior prioridade no estado pronta.
5. Tarefa 2 encerra a sua execução e o funcionamento do escalonador é passado para a tarefa 1.

Figura 3.4: Funcionamento de um escalonador preemptivo baseado na prioridade [**?**]

Para as co-rotinas, o escalonador funciona de forma cooperativa e baseada na prioridade. Assim, a co-rotina em execução é quem decide o momento da sua interrupção e, em seguida, o sistema deve chamar o escalonador através de uma funcionalidade específica para decidir qual será a próxima co-rotina que irá entrar em execução. A escolha da próxima co-rotina a ser executada também é baseada na maior prioridade, assim como ocorre com as tarefas. Um exemplo do funcionamento do escalonador pode ser visto na seção 3.3.4.

3.1.4 Bibliotecas

Para disponibilizar as características discutidas nesta seção, o FreeRTOS contém uma biblioteca de tipos e funções organizada da seguinte forma: criação de tarefas, controle de tarefas, utilidades de tarefas, controle do kernel e co-rotinas. A seguir, tem-se em detalhes a descrição de cada uma dessas bibliotecas junto com os tipos e funcionalidades que as compõem.

Criação de Tarefas

Essa parte é responsável pela entidade tarefa e sua criação. Nela, estão presentes um tipo, responsável por representar uma tarefa do sistema, e duas funcionalidades, uma para a criação e outra para a remoção de tarefas do sistema. Em seguida, tem-se o tipo e as funcionalidades que compõem a biblioteca em questão.

- **xTaskCreate**: Cria uma nova tarefa para o sistema.
- **vTaskDelete**: Remove uma tarefa do sistema[3].
- **xTaskHandle**: Tipo pelo qual uma tarefa e referenciada. Por exemplo, quando uma tarefa é criada através do método *xTaskCreate*, este retorna uma referência para nova tarefa através do tipo *xTaskHandle*.

Controle de tarefas

A biblioteca de controle de tarefas realiza operações sobre as tarefas do sistema. Ela disponibiliza funcionalidades capazes de bloquear, suspender, reativar uma tarefa do sistema, informar e alterar a prioridade de uma tarefa no sistema. A lista das principais funcionalidades presentes nessa biblioteca pode ser vista a seguir:

- **vTaskDelay**: Bloqueia uma tarefa por um determinado tempo. Nessa funcionalidade, para calcular o tempo que a tarefa deve permanecer bloqueada, será levado em consideração o instante da chamada à funcionalidade. Devido a isso, essa funcionalidade não é recomendada para a criação de tarefas cíclicas, pois o instante em que ela é chamada pode variar a cada execução da tarefa, por causa das interrupções que uma tarefa pode sofrer.
- **vTaskDelayUntil**: Bloqueia uma tarefa por um determinado tempo. Essa funcionalidade, diferente da *vTaskDelay*, calcula o tempo que a tarefa deve permanecer bloqueada com base no instante do último desbloqueio da tarefa. Assim, se ocorrer uma interrupção no momento da chamada é funcionalidade, o instante que a tarefa foi desbloqueada não ira mudar. Com isso, esse método torna-se recomendável para a criação de tarefas cíclicas.
- **uxTaskPriorityGet**: Informa a prioridade de uma determinada tarefa.
- **vTaskPrioritySet**: Muda a prioridade de uma determinada tarefa.
- **vTaskSuspend**: Coloca uma determinda tarefa no estado suspensa.
- **vTaskResume**: Coloca uma determinada tarefa suspensa no estado pronta.
- **xTaskResumeFromISR** - Funcionalidade usada pelo tratamento de interrupções do sistema. Ela coloca uma determinada tarefa suspensa para o estado pronta (usada no tratamento de interrupções).

[3] A memória alocada pela tarefa será liberada somente quando a tarefa ociosa entrar em execução(seção 3.1.1)

Utilitários de tarefas

Através dessa parte, o FreeRTOS disponibiliza, para o usuário, informações importantes a respeito das tarefas e do escalonador de tarefas. Nela, estão presentes funcionalidades capazes de retornar uma referência para a atual tarefa em execução, retornar o tempo de funcionamento e o estado do escalonador e retornar o número de tarefas que estão sendo gerenciadas pelo sistema. Uma lista das principais funcionalidades dessa biblioteca é encontrada a seguir:

- **xTaskGetCurrentTaskHandle**: Retorna uma referência para a atual tarefa em execução.
- **uxTaskGetStackHighWaterMark**: Retorna a quantidade de espaço restante na pilha de uma tarefa.
- **xTaskGetTickCount**: Retorna o tempo decorrido desde a inicialização do escalonador.
- **xTaskGetSchedulerState**: Retorna o estado do escalonador.
- **uxTaskGetNumberOfTasks**: Retorna o número de tarefas do sistema.
- **TaskCallApplicationTaskHook**:Chama a função gancho, função associado para preceder a execução de uma tarefa, de uma determinada tarefa.
- **TaskSetApplicationTag**: Associa uma 'tag' a uma tarefa. Essa tag será utilizada principalmente pelas funcionalidades de rastreamento do sistema. Entretanto, é possível usar essa 'tag' para associar uma função gancho a uma tarefa. Essa função é executada através da chamada é funcionalidade *TaskCallApplicationTaskHook*, informando a tarefa associada.

Controle do Escalonador

Nessa biblioteca estão presentes as funcionalidades responsáveis por controlar as atividades do escalonador de tarefas. Nela, encontram-se as funcionalidades que iniciam, finalizam, suspendem e reativam as atividades do escalonador. As principais funcionalidades presente nessa biblioteca são:

- **vTaskStartScheduler**: Inicia as atividades do escalonador, ou seja, inicializa o sistema.
- **vTaskEndScheduler**: Encerra as atividades do escalonador, ou seja, finaliza o sistema.
- **vTaskSuspendAll**: Suspende as atividades do escalonador.
- **xTaskResumeAll**: Reativa o escalonador quando o mesmo está suspenso.
- **taskYIELD**: Força a troca de contexto[4] entre tarefas.
- **taskENTER_CRITICAL**: Indica o início de uma região crítica, desabilita temporariamente a característica de preempção do escalonador impedindo que a tarefa em execução seja interrompida por outra.

[4]Troca de contexto é a operação na qual a tarefa em execução é trocada por outra. Para que a troca de contexto sejá realizada, deve existir uma tarefa de prioridade igual é da tarefa em execução

- **taskEXIT_CRITICAL**: Indica o final de uma região crítica, permitindo que ocorrá novamente o escalonamento preemptivo.
- **taskDISABLE_INTERRUPTS**: Desabilita as interrupções do microcontrolador.
- **taskENABLE_INTERRUPTS**: Habilita as interrupções do microcontrolador.

Co-rotina

A última biblioteca do serviço de gerenciamento de tarefa e co-rotina é a biblioteca co-rotina. Nela, estão presentes as funcionalidades e tipos responsáveis por criar e gerenciar o elemento co-rotina. A lista completa das funcionalidades presentes nessa biblioteca pode ser vista a seguir:

- **xCoRoutineHandle**: Tipo responsável por representar uma co-rotina.
- **xCoRoutineCreate**: Cria uma nova co-rotina no sistema.
- **crDELAY**: Bloquea uma co-rotina durante uma determinada quantidade de tempo.
- **crQUEUE_SEND**: Envia uma mensagem para uma fila através de uma co-rotina.
- **crQUEUE_RECEIVE**: Recebe uma mensagem de uma fila através de uma co-rotina.
- **crQUEUE_SEND_FROM_ISR**: Envia uma mensagem para uma fila, através de uma co-rotina responsável por tratar uma interrupção.
- **crQUEUE_RECEIVE_FROM_ISR**: Recebe uma mensagem de uma fila, através de uma co-rotina responsável por tratar uma interrupção.
- **vCoRoutineSchedule**: Chama o escalonador para escolher e colocar em execução a co-rotina de maior prioridade entre as co-rotinas de estado pronto.

3.2 Comunicação e sincronização entre tarefas

Frequentemente tarefas necessitam comunicar entre si. Por exemplo, a tarefa *A* depende da leitura do teclado, feita pela tarefa *B*, para disponibilizar em uma tela as teclas digitadas pelo usuário. Para que essa comunicação possa ser estruturada e sem interrupções, os sistemas operacionais possuem mecanismos específicos de comunicações.

A maioria dos sistemas operacionais oferece vários tipos de comunicação entre as tarefas. Geralmente esses tipos são: tarefas trocando informações entre si; tarefas utilizando, de forma sincronizada, o mesmo recurso; tarefas dependentes dos resultados produzidos por outras.

No FreeRTOS, como nos demais sistemas operacionais, os mecanismos responsáveis por realizar a comunicação entre as tarefas são as filas de mensagens, os semáforos e o mutexes (*Mutal Exclusion*). Para entender melhor como funciona essa comunicação, cada um desses mecanismos será detalhado a seguir.

3.2.1 Fila de Mensagens

Filas de mensagens são estruturas primitivas de comunicação entre tarefas. Elas funcionam como um túnel, através do qual tarefas enviam e recebem mensagem (figura 3.5). Assim, quando uma tarefa necessita comunicar-se com outra, ela envia uma mensagem para o túnel para que a outra tarefa possa ler sua mensagem [**?**].

Figura 3.5: Funcionamento de uma fila de mensagens.

No FreeRTOS, uma fila de mensagens é formada por:

- A lista das mensagens na fila;
- A lista de tarefas que aguardam para enviar uma mensagem para a fila;
- A lista de tarefas que aguardam pela chegada de uma mensagem na fila;
- Uma variável que indica o tamanho das mensagens da fila; e
- Uma variável responsável por indicar o tamanho máximo da fila, quantidade de mensagens que podem ser armazenadas pela fila.

O funcionamento de uma fila de mensagens no FreeRTOS ocorre da seguinte forma. Primeiro a tarefa remetente envia uma mensagem para a fila e, em seguida, a tarefa receptora retira a mensagem da fila. Entretanto, se, no momento do envio da mensagem, a fila estiver cheia a tarefa remetente é bloqueada e colocada na lista de tarefas que aguardam para enviar uma mensagem para a fila. O mesmo ocorre quando a tarefa receptora tenta receber uma mensagem de uma fila vazia.

A retirada de uma tarefa das listas de espera de uma fila é feita levando em consideração a prioridade das tarefas da lista. Assim, quando uma mensagem é retirada de uma fila cheia, a lista de tarefas que aguardam para enviar uma mensagem para a fila é percorrida e a tarefa de maior prioridade é retirada da fila, sendo consequentemente desbloqueada. Fato parecido ocorre com a fila de tarefas bloqueadas em leitura quando uma mensagem chega a uma fila vazia.

No momento de enviar uma mensagem para uma fila, uma tarefa pode especificar o tempo máximo que ela deve permanecer bloqueada, aguardando para enviar a mensagem. Assim como, ao solicitar uma mensagem para a fila, uma tarefa também pode definir o tempo máximo que ela pode ficar bloqueada, esperando pela chegada de uma mensagem na fila. As funcionalidades que tornam possível essas características serão demonstradas na seção 3.2.4.

3.2.2 Semáforo

Os semáforos são mecanismos usados para realizar a sincronização entre tarefas. Eles funcionam como uma chave de pré-condição para uma tarefa executar

uma operação sincronizada ou acessar um recurso compartilhado. Assim, antes de executar tal ação, a tarefa deve solicitar o semáforo responsável pela guarda da ação. Caso o semáforo esteja disponível, a tarefa realiza a ação, caso contrário, a tarefa é bloqueada até que o semáforo seja liberado.

O FreeRTOS disponibiliza dois tipos de semáforos para o usuário, o semáforo binário e o semáforo com contador. A diferença entre esses dois tipos de semáforo está no número de tarefas que podem reter o semáforo ao mesmo tempo. No semáforo binário é possível apenas uma tarefa manter o semáforo. Entretanto, no semáforo com contador, existe um número fixo de tarefa (maior ou igual a um) que podem reter o semáforo.

Para controlar o acesso de várias tarefas ao semáforo com contador, ele possui uma variável denominada contador, cujo valor é definido no momento da criação do semáforo. Assim, seu funcionamento ocorre como demonstra a figura 3.6, para cada tarefa que retêm o semáforo, o contador é decrementado e, para cada tarefa que libera o semáforo, o contador é incrementado. Com isso, o semáforo estará indisponível quando o valor do contador for igual a zero e seu valor não poderá ultrapassar o número definido inicialmente.

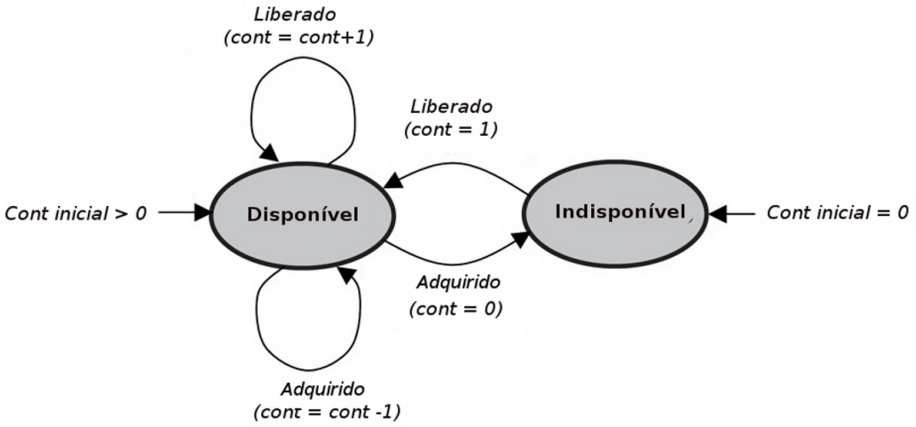


Figura 3.6: Diagrama de estado do semáforo com contador.

No FreeRTOS, os semáforos são implementados através de uma fila de mensagens que informa o estado do semáforo através da situação da fila. Assim, quando a fila estiver vazia, indica que o semáforo não poderá ser retido e, nas demais situações da fila, indica que o semáforo está liberado. Com isso, para representar-se um semáforo binário, é criada uma fila de capacidade um e, para representar um semáforo com contador, é criada uma fila de capacidade igual ao valor inicial do contador do semáforo.

3.2.3 Mutex

Mutexes são estruturas parecidas com os semáforos binários. A única diferença entre os dois é que o mutex implementa o mecanismo de herança de pri-

oridade, o qual impede que uma tarefa, de maior prioridade, fique bloqueada a espera de um semáforo ocupado por outra tarefa, de menor prioridade, causando assim uma situação de bloqueio por inversão de prioridade.

O mecanismo de herança de prioridade funciona como demonstra a figura 3.7. Quando uma tarefa solicita o mutex, ele primeiro verifica se a tarefa solicitante possui prioridade maior que a tarefa com o semáforo. Caso afirmativo, a tarefa que retêm o semáforo tem, momentaneamente, a sua prioridade elevada, para que assim ela possa realizar as suas funções sem interrupções e, consequentemente, liberar o semáforo mais rapidamente. Um detalhe interessante desse elemento é que ele utiliza as mesmas funcionalidades do semáforo para reter e liberar o mutex. Essas funcionalidades serão explicadas na seção 3.2.4.

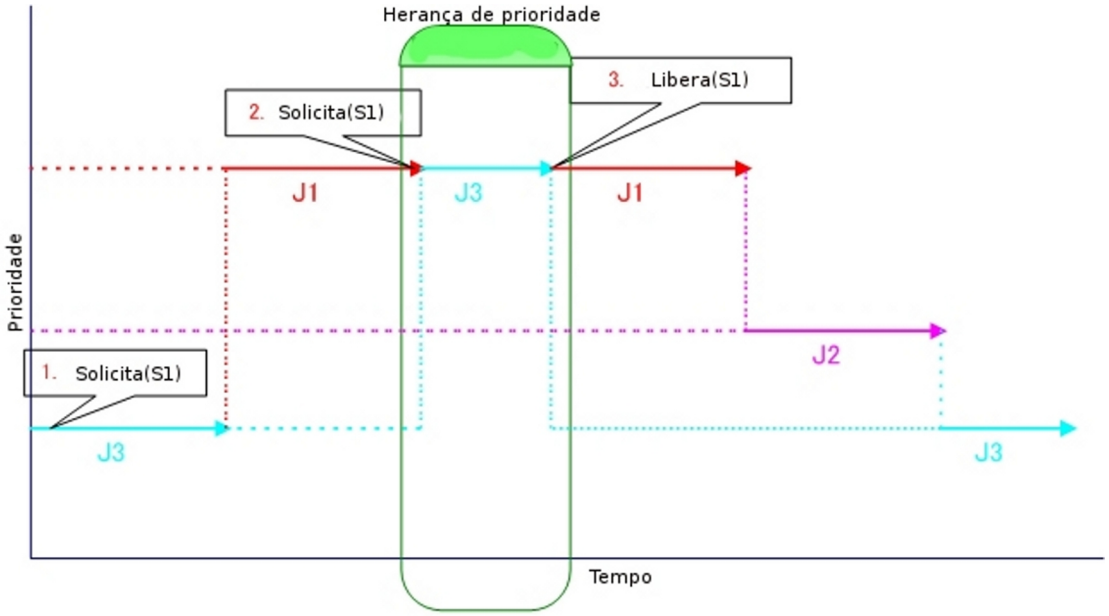


Figura 3.7: Funcionamento do mecanismo de herança de prioridade.

3.2.4 Biblioteca

Para disponibilizar as características de comunicação e sincronização entre tarefas, o FreeRTOS dispõe de um conjunto de funcionalidades e tipos agrupados em duas bibliotecas: gerenciamento de fila de mensagens e semáforo/mutex. Juntas essas bibliotecas possuem vinte e quatro funcionalidades, das quais as principais serão listadas nas seções seguintes.

Gerenciamente de fila de Mensagens

O conjunto de funcionalidades de gerenciamento de uma fila de mensagens é responsável pela criação e utilização da entidade fila de mensagens. Nele estão

presentes funcionalidades responsáveis por instanciar e remover uma fila do sistema e funcionalidades que enviam e recebem mensagens de uma fila do sistema. A seguir, tem a lista das funcionalidades mais relevantes dessa biblioteca.

- **xQueueCreate**:Cria uma nova fila de mensagens no sistema.
- **vQueueDelete**: Remove uma fila de mensagens do sistema.
- **xQueueSend**: Envia uma mensagem para uma fila, sem uma determinada extermidade.
- **xQueueSendToBack**: Envia uma mensagem para o final de uma fila.
- **xQueueSendToFront**: Envia uma mensagem para o início de uma fila.
- **xQueueReceive**:Retira uma mensagem de uma fila.
- **xQueuePeek**: Lê uma mensagem de uma fila, sem remové-la.
- **xQueueSendFromISR**: Manda uma mensagem para uma fila, a partir de uma tarefa de tratamento de interrupção.
- **xQueueSendToBackFromISR**: Manda uma mensagem para o final de uma fila, a partir de uma tarefa de tratamento de interrupção.
- **xQueueSendToFrontFromISR**: Manda uma mensagem para o início de uma fila, a partir de uma tarefa de tratamento de interrupção.
- **xQueueReceiveFromISR** : Lê/Retira uma mensagem de uma fila, a partir de uma tarefa de tratamento de interrupção.

Semáforo/Mutex

Na biblioteca de semáforo e mutex estão implementadas, junto com as suas funcionalidades, as estruturas de sincronização entre tarefas semáforo e mutex. Assim, nesta biblioteca estão presentes funcionalidades que criam e removem semáforos e mutex do sistema, como também funcionalidades utilizadas para solicitar e liberar um semáforo ou mutex. As principais funcionalidades desta biblioteca podem ser vistas a seguir:

- **vSemaphoreCreateBinary**: Cria um semáforo binário.
- **vSemaphoreCreateCounting**: Cria um semáforo com contador.
- **xSemaphoreCreateMutex**: Funcionalidade usada para criar um mutex.
- **xSemaphoreTake**: Retém um semáforo ou um mutex.
- **xSemaphoreGive**: Funcionalidade usada para liberar um semáforo ou um mutex retido.
- **xSemaphoreGiveFromISR**: Libera um semáforo binário ou com contador, a partir de uma tarefa de tratamento de interrupção (não deve ser utilizada para mutex).

3.3 Utilização prática dos elementos FreeRTOS

Para construir uma aplicação de tempo real utilizando o FreeRTOS, o desenvolvedor deve seguir determinadas restrições impostas pelo sistema. A maioria destas restrições são parâmetros de configuração e modelos para a criação

dos elementos do sistema. Assim, com o intuito de ajudar o desenvolvedor a criar suas primeiras aplicações, o FreeRTOS disponibilizou, junto com seu código fonte, aplicações exemplos classificadas por plataformas. Desse modo, essas aplicações exemplos podem ser utilizadas como ponto de partida na criação de novos projetos.

Entretanto, a criação e análise de uma nova aplicação no FreeRTOS é uma atividade que necessita de maior conhecimento sobre as suas funcionalidades, fugindo assim do objetivo geral desse capítulo, que é proporcionar uma breve introdução ao FreeRTOS, demonstrando seus principais conceitos, funcionalidades e características. Com isso, para um melhor entendimento das explicações apresentadas neste capítulo, será demonstrada, nas seções seguintes, de forma didática, a utilização das principais entidades aqui discutidas, tarefa, co-rotinas, fila de mensagens e semáforos.

3.3.1 Utilização da entidade tarefa

A tarefa é a parte mais importante de uma aplicação. Nesta são colocadas as ações responsáveis pelo funcionamento da aplicação[**?**]. No FreeRTOS, as ações realizadas pelas tarefas são colocadas dentro de rotinas, as quais devem seguir uma estrutura pré-definida, demonstrada pela figura 3.8. Nela, tem-se que uma rotina deve ser formada inicialmente por um cabeçalho com o seu nome e seguido de uma lista de parâmetros utilizados por ela. Em seguida, tem-se o código que realiza as finalidades da tarefa, no qual um laço infinito é colocado para abrigar a parte repetitiva desse código. Assim, a atividade de uma tarefa nunca termina, ficando sob o controle do escalonador.

```
void functionName( void *vParameters )
{
      for( ;; )
      {
          -- Task application code here. --
      }
}
```

Figura 3.8: Estrutura da rotina de uma tarefa.

Um exemplo concreto da criação de uma tarefa pode ser visto na figura 3.9. Nela, tem-se a rotina *cyclicalTasks*, que utiliza a funcionalidade *vTaskDelayUntil* (seção 3.1.4) para bloquear a execução da tarefa em intervalos iguais de tempo. Essa funcionalidade possui como parâmetros, respectivamente, o último tempo que a tarefa foi reativada do estado suspensa e o período que a tarefa deve permanecer bloqueada. Após isso, a tarefa é criada no sistema junto com sua prioridade, pilha de contexto e nome. Essa operação é feita através da funcionalidade *xTaskCreate* que possui como parâmetros os seguintes argumentos:

cyclicalTasks : Ponteiro para a rotina que deve ser executada pela tarefa;
"cyclicalTasks" : Nome da tarefa utilizada nos arquivos de log do sistema;
STACK_SIZE : Tamanho da pilha de execução da função especificado de acordo com o número de variáveis declaradas na rotina da função;
pvParameters : Lista de valores dos parâmetros de entrada da rotina da função;
TASK_PRIORITY : Prioridade da tarefa;
cyclicalTasksHandle : Gancho de retorno da tarefa criada.

```
void cyclicalTasks( void * pvParameters ){
    portTickType xLastWakeTime;
    const portTickType xFrequency = 10;
    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount();
    for( ;; ){
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );
        // Perform action here.
    }
}

xTaskHandle cyclicalTasksHandle;

xTaskCreate( cyclicalTask, "cyclicalTasks", STACK_SIZE,
                ( void * ) pvParameters, TASK_PRIORITY, &cyclicalTasksHandle);

vTaskStartScheduler();
```

Figura 3.9: Aplicação formada por uma tarefa cíclica.

Para finalizar o exemplo da figura 3.9, após ser criada a tarefa, o escalonador do sistema deve ser iniciado e com ele a aplicação. Essa operação é feita pela a funcionalidade *vTaskStartScheduler()*, localizada no final do código.

3.3.2 Utilização da fila de mensagens

A utilização de uma fila de mensagens é resumidamente demonstrada na aplicação da figura 3.10. Nela, inicialmente tem-se a estrutura *AMessage*, que define o tipo da mensagem que será utilizada. Em seguida, através do método *xQueueCreate*, é criada uma fila de mensagens que será referenciada pela variável *xQueue*, do tipo *xQueueHandle*. Para isso, o método *xQueueCreate* recebe como parâmetros, respectivamente, a quantidade de mensagens que a fila pode armazenar e o tamanho da mensagens manuseadas por ela.

Estabelecida a fila de mensagens, é necessário agora definir as tarefas que irão enviar e receber mensagens da mesma. Na figura 3.10, estão presentes apenas

```
struct AMessage {
    portCHAR ucMessageID;
    portCHAR ucData[ 20 ];
}xMessage;

xQueueHandle xQueue;
//Create a queue capable of containing 10 pointers to AMessage structures.
xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
// Task to create a queue and post a value.
void sendTask( void *pvParameters ) {
struct AMessage *pxMessage;

    if( xQueue == 0 ) {
        // Failed to create the queue.
    }else{
        // Send a pointer to a struct AMessage object. Don't block if the
        // queue is already full.
        pxMessage = & xMessage;
        xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );
    }
// ... Rest of task code.

}

// Task to receive from the queue.
void receiveTask( void *pvParameters ) {
struct AMessage *pxRxedMessage;

    if( xQueue != 0 ) {
        // Receive a message on the created queue. Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) ) {
            // pcRxedMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }
    // ... Rest of task code.
}
```

Figura 3.10: Aplicação que utiliza uma fila de mensagens.

as rotinas de cada uma dessas tarefas, sendo que a explicação completa de como é criada uma tarefa foi demonstrada na seção 3.3.1. A seguir, tem a explanação sobre cada uma dessas rotinas.

Para enviar uma mensagem para a fila *xQueue*, a rotina *sendTask* utiliza-se da

funcionalidade *xQueueSend*. Essa funcionalidade possui como parâmetros a fila para qual a mensagem será enviada, a mensagem que será enviada para a fila e o tempo máximo que a tarefa poderá ficar bloqueada aguardando para enviar a mensagem.

Por último, na rotina *receiveTask*, uma mensagem é retirada da fila *xQueue* através da funcionalidade *xQueueReceive*. Para isso, ela utiliza como argumentos, respectivamente, a fila onde será retirada a mensagem, o endereço onde a mensagem será armazenada, e o tempo máximo que a tarefa pode ficar esperando pela fila. Como retorno, essa funcionalidade informa se a mensagem foi retirada da fila com sucesso ou não, sendo, com isso, utilizada como guarda para o código que deve ser executado após a retirada da mensagem.

3.3.3 Utilização do semáforo

Para a construção de uma aplicação que se utiliza do semáforo são necessárias basicamente três funcionalidades da biblioteca de semáforos, *vSemaphoreCreateBinary*, *xSemaphoreTake* e *xSemaphoreGive*. A primeira funcionalidade cria o semáforo e as demais solicitam e liberam o semáforo, respectivamente.

Um exemplo de uma aplicação que utiliza um semáforo para controlar o acesso de um recurso compartilhado pode ser visto na figura 3.11. Nela, inicialmente é criada a variável *xSemaphore* para armazenar uma referência ao novo semáforo. Em seguida, o método *vSemaphoreCreateBinary* é usado para criar o novo semáforo e retornar uma referência para o mesmo.

Após a criação do semáforo, a rotina da tarefa que utilizará o recurso compartilhado é desenvolvida. Nela, o código que acessará tal recurso está protegido pela segunda condição *if*, a qual recebe o retorno do método *xSemaphoreTake*, informando se o semáforo foi retido ou não. Ao final da rotina, o semáforo é liberado pelo método *xSemaphoreGive*, permitindo que outra tarefa possa retê-lo e usar o recurso compartilhado.

A utilização do mutex é bem parecida com a do semáforo binário. A diferença, para o usuário, entre os dois mecanismos está apenas no método de criação da entidade, *xSemaphoreCreateMutex*. As formas e os métodos para solicitar e liberar o mutex são os mesmos do semáforo, *xSemaphoreTake* e *xSemaphoreGive*. Com isso, para transformar a aplicação da figura 3.11 de semáforo para mutex basta apenas trocar o método *vSemaphoreCreateBinary* por *xSemaphoreCreateMutex*.

3.3.4 Utilização das co-rotinas

As co-rotinas, assim como as tarefas, possuem as ações responsáveis pelas atividades da aplicação. Essas ações são colocadas dentro das rotinas executadas pelas co-rotinas, as quais devem seguir o modelo demonstrado na figura 3.12. Nele, a rotina deve possuir um nome, seguido de dois parâmetros: *xHandle*, uma referência para a co-rotina que será utilizado pelas funcionalidades de controle da co-rotina; e *uxIndex*, que é usado opcionalmente no desenvolvimento do código

```
xSemaphoreHandle xSemaphore = NULL;


// Create the semaphore to guard a shared resource. As we are using
// the semaphore for mutual exclusion we create a mutex semaphore
// rather than a binary semaphore.
xSemaphore = xvSemaphoreCreateBinary();

// A task that uses the semaphore.
void semaphoreTask( void * pvParameters ) {
    // ... Do other things.
    if( xSemaphore != NULL ) {
        // See if we can obtain the semaphore. If the semaphore is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTake( xSemaphore, ( portTickType ) 10 ) == pdTRUE ) {
            // We were able to obtain the semaphore and can now access the
            // shared resource.
            // We have finished accessing the shared resource. Release the
            // semaphore.
            xSemaphoreGive( xSemaphore );
        } else
            // We could not obtain the semaphore and can therefore not access
            // the shared resource safely
       {
    }
}
```

Figura 3.11: Aplicação que demonstra a utilização de um semáforo

da co-rotina. Por último, o código de execução da co-rotina deve ser iniciado com a função *crSTART(xHandle)* e finalizado pela função *crEND()*.

```
void vACoRoutineFunction(xCoRoutineHandle xHandle,
unsigned portBASE_TYPE uxIndex ){
        crSTART( xHandle );
        for( ;; )
        {
            -- Co-routine application code here. --
        }
        crEND();
    }
```

Figura 3.12: Modela da rotina de execução de uma co-rotina.

Um exemplo completo da utilização de uma co-rotina pode ser vito na fi-

gura 3.13. Nesse é criada uma co-rotina que, através da rotina *vFlashCoRoutine*, controla o funcionamento de um LED. Para isso ela interrompe a sua execução durante um tempo de dez ticks (unidade de tempo do FreeRTOS), utilizando a funcionalidade *crDELAY*, que tem como argumentos o gancho da co-rotina e o tempo de bloqueio da co-rotina. Após isso, a função de controle do LED *vParTestToggleLED* é chamada. Observa-se, nesse exemplo, a utilização dos parâmetros da rotina como argumentos para a funcionalidade *crDELAY*.

Com a criação da rotina utilizada pela co-rotina, resta apenas criar a co-rotina no sistema. Esse trabalho é feito através da funcionalidade *xCoRoutineCreate*, que recebe como parâmetros respectivamente a rotina que será associada à co-rotina, a prioridade da co-rotina e o valor do parâmetro *uxIndex* da rotina associada à co-rotina.

Por fim, devido à política de escalonamento cooperativo das co-rotinas, ao final da aplicação da figura 3.13 é colocada na rotina *vApplicationIdleHook* (rotina executada pela tarefa ociosa, seção 3.1.1) a funcionalidade *vCoRoutineSchedule* que chama o escalonador para realizar a troca da co-rotina em execução. Assim, sempre que a co-rotina em execução é bloqueada, a tarefa ociosa entra em ação chamando o escalonador para fazer a troca de co-rotinas.

Através desse capítulo, pode-se perceber que o *microkernel* FreeRTOS é formado por abstrações de hardware que dão suporte para a construção de novas aplicações. Essas aplicações utilizam tais abstrações por intermédio das funções das bibliotecas disponibilizadas pelo sistema. Cada função realiza alterações no comportamento interno do núcleo e é através dessa visão que o FreeRTOS será especificado nesse trabalho, dando ênfase nas alterações geradas pela API.

```
...
void main( void )    {
// This time i is passed in as the index.
xCoRoutineCreate( vFlashCoRoutine, PRIORITY_0, 10 );

    // NOTE: Tasks can also be created here!
// Start the scheduler.
    vTaskStartScheduler();
}

void vFlashCoRoutine( xCoRoutineHandle xHandle,
unsigned portBASE_TYPE uxIndex ){
// Co-routines must start with a call to crSTART().
crSTART( xHandle );
        for( ;; ){
            crDELAY( xHandle, uxIndex);
            vParTestToggleLED;
        }
// Co-routines must end with a call to crEND().
crEND();

}

void vApplicationIdleHook( void ){
        vCoRoutineSchedule( void );
}
```

Figura 3.13: Aplicação que demonstra a utilização de uma co-rotina.

Capítulo 4

O método B

Métodos Formais provêem abordagens formais para a especificação e construção de sistemas computacionais. Eles utilizam conceitos matemáticos sólidos como lógica de primeira ordem e teorias dos conjuntos para a criação e verificação de sistemas consistentes, seguros e sem ambiguidades. Devido aos seus rigorosos métodos de construção, a sua principal utilização tem sido na criação de sistemas críticos para as indústrias de aeronáutica, viação férrea, equipamentos médicos e empresas que movimentam grandes quantidades monetárias, como os bancos[**?**].

Segundo a Honeywell, uma empresa que desenvolve sistemas para aeronaves, a utilização de métodos formais no processo de desenvolvimento provê várias vantagens, entre eles estão [**?**]:

- A produção mensurada pela corretude: métodos formais provêem uma forma objetiva de mensurar a corretude do sistema;
- Antecipação na detecção de erros: métodos formais são usados previamente em projetos de artefatos do sistema, permitindo assim a sua verificação e com isso a detecção antecipada de possíveis erros do projeto;
- Garantia da corretude: através de mecanismo de verificação formal é possível provar que sistema funcionará de forma coerente com a sua especificação inicial.

O método B[**?**] é uma abordagem formal usada para especificar e construir sistemas computacionais seguros. Seu criador, Jean-Raymond Abrial, junto com a colaboração de outros pesquisadores da universidade de Oxford, procurou reunir no método B vários conceitos presentes nos demais métodos formais. Entre esses conceitos, destacam-se as pré e pós condições (seção 4.2.2), o desenvolvimento incremental através de refinamentos (seção 4.4) e a modularização da especificação. A seguir, tem a explicação de como é feito o desenvolvimento de sistemas utilizando o método B.

4.1 Etapas do desenvolvimento em B

O processo de desenvolvimento através do método B inicia com a criação de um módulo, que define em alto nível um modelo funcional do sistema. Em B,

esses módulos são denominados de Máquinas Abstratas (*MACHINE*). Nessa fase de modelagem, técnicas semi-formais como UML podem ser utilizadas e, em seguida, transformadas para a notação formal do método B. Após a criação dos módulos, esses são analisados estaticamente para verificar se são coerentes e implementáveis.

Uma vez estabelecido o modelo abstrato inicial do sistema, o método B permite que sejam construídos módulos mais concretos do sistema, denominados refinamentos. Mais especificamente, refinamentos correspondem a uma decisão de projeto, na qual partes da especificação abstrata do sistema devem ser modeladas em um nível mais concreto. Assim, um refinamento deve necessariamente estar relacionado com o módulo abstrato imediatamente anterior. Como ocorre na criação das máquinas abstratas, um refinamento também é passível de uma análise estática, na qual é verificada a relação entre o refinamento e o seu nível abstrato anterior.

Devido à técnica de refinamentos, o desenvolvimento de sistemas utilizando o método B pode chegar a um nível de abstração semelhante aos das linguagens de programação imperativas e sequenciais. Para isso, sucessivos refinamentos devem ser desenvolvidos até a especificação chegar a um último nível de refinamento denominado implementação (*IMPLEMENTATION*). Nesse nível, a linguagem utilizada, chamada B0, é um formalismo algorítmico passível de ser sintetizado em linguagens de programação como C, Java e JavaCard.

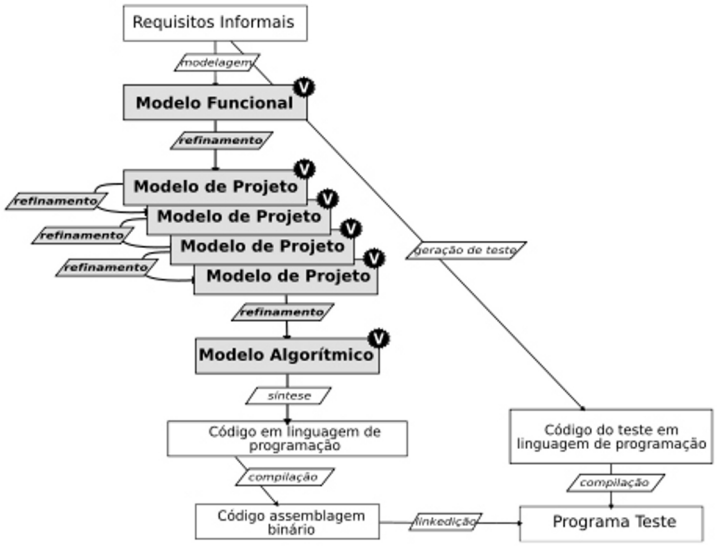


Figura 4.1: Etapas do desenvolvimento de sistema através do método B.

O desenvolvimento de sistemas utilizando o método B é feito como demonstra a figura 4.1. Nela, os requisitos do sistema são inicialmente especificados em um alto nível de abstração e, após sucessivos refinamentos, um nível algorítmico do sistema é alcançado. Em seguida, essa especificação é sintetizada para um código de linguagem de programação, para o qual é possível, a partir da especificação

funcional inicial, gerar testes para validar a sua correta transformação.

Atualmente, o desenvolvimento de sistemas utilizando o método B pode ser apoiado por diversas ferramentas com funcionalidades que vão da análise estática da especificação até a geração de código em linguagens de programação. Uma das mais famosas e completas ferramentas de apoio ao desenvolvimento de sistemas utilizando o método B é o AtelierB [**?**]. Nela, é possível, além da análise sintática e estática da especificação, gerenciar projetos e controlar as dependências entre os vários módulos que constituem uma especificação. Devido as suas vastas funcionalidades e popularidade, o AtelierB será adotado como ferramenta padrão desse trabalho.

Um exemplo de sistema desenvolvidos através da abordagem B, com o apoio da ferramenta Atelier B, é o controle de porta de plataforma em uma linha de metrô de Paris, desenvolvido pela Clearsy[1], empresa especialista em sistemas críticos.[**?**].

4.2 Máquina abstrata

A base do método B está na notação de máquina abstrata (em inglês: *Abstract Machine Notation* - AMN), a qual disponibiliza um framework comum para a especificação, construção e verificação estática de sistemas. Em outras palavras, a AMN é uma linguagem de especificação de sistemas formada por módulos básicos de construção chamados de máquina abstrata ou simplesmente máquina.

Cada máquina abstrata é composta por diferentes seções, sendo que cada seção é responsável por definir um aspecto da especificação do sistema como: parâmetros, tipos, constantes, variáveis de estado, estados iniciais e transições do sistema. Por exemplo, a figura 4.2 contêm uma máquina abstrata, chamada *Kernel*, a qual especifica um sistema que permite incluir e excluir tarefas até o limite de 10. Essa máquina possui as seguintes seções:

MACHINE é onde se inicia o código da máquina abstrata. Nela, é identificado a natureza e o nome do módulo, seguido opcionalmente por um ou mais parâmetros de máquina, os quais são separados por vírgula e limitados por parênteses;

SETS introduz um novo tipo de entidade, como é o caso de *TASK* no exemplo em questão. Nesse momento, nenhum detalhe é fornecido quanto a maneira como essa entidade será implementada;

VARIABLES informa o nome das diferentes variáveis que compõem o estado da máquina. No exemplo da figura 4.2, há apenas uma variável de estado, *tasks*;

INVARIANT especifica o tipo das variáveis de estado e os estados válidos do sistema. No exemplo em questão, a variável *tasks* é um conjunto de até 10 elementos do tipo *TASK*. A caracterização lógica do conjunto dos estados válidos é uma das atividades mais importantes da especificação;

[1] www.clearsy.com

INITIALISATION identifica quais são os possíveis estados iniciais do sistema. No caso da figura 4.2, *tasks* é inicializado como um o conjunto vazio; e

OPERATIONS determina os diferentes tipos de eventos que o sistema pode sofrer. No exemplo demonstrado, tem-se operações para adicionar e eliminar um elemento de *tasks*. Uma operação pode ter parâmetros, resultados e alterar o valor de variáveis de estado. Um ponto importante encontrado nas operações são as pré-condições, as quais são condições que devem ser satisfeitas para que a operação seja realizada com sucesso.

MACHINE
Kernel
SETS
TASK
VARIABLES
tasks
INVARIANT
$tasks \in \mathrm{P}(TASK) \wedge$
$\mathbf{card}(tasks) \leq 10$
INITIALISATION
$tasks := \emptyset$

OPERATIONS

$task\_add(task) =$
PRE
$task \in TASK \wedge$
$task \notin tasks \wedge$
$\mathbf{card}(tasks) < 10$
THEN
$tasks := tasks \cup \{task\}$
END;

...
$task\_delete(task) =$
PRE
$task \in tasks$
THEN
$tasks := tasks - \{task\}$
END
END

Figura 4.2: Máquina abstrata de tarefas.

Para uma melhor compreensão, a especificação de sistemas através das máquinas abstratas será resumidamente dividida em duas partes principais. Na primeira parte, serão colocadas informações a respeito dos estados da máquina, suas variáveis e restrições. Na segunda parte, será especificado o comportamento da máquina, ou seja, a sua parte dinâmica, como a inicialização e as operações. Essas duas partes serão melhor discutidas a seguir.

4.2.1 Especificação do estado da máquina

Nessa parte, são determinados os estados que uma máquina pode assumir. Esses estados são definidos através das variáveis e dos seus invariantes (**INVARIANT**). As variáveis são diferentes elementos que compõem o estado do sistema. Os invariantes são expressões lógicas que determinam os valores que as variáveis podem assumir. Assim, uma especificação só pode definir o correto funcionamento da máquina, quando ela encontra-se em um estado válido, nada sendo especificado para os demais casos.

O estado de uma máquina é especificado por meio de lógica dos predicados, teoria dos conjuntos e aritmética dos inteiros, permitindo com isso uma análise estática da máquina através das expressões lógicas geradas a partir de sua especificação. No exemplo da figura 4.2, o estado da máquina *kernel* foi especificado através da variável *tasks*, sendo $tasks \in \mathrm{P}(TASK)$ e $\mathbf{card}(tasks) \leq 10$, o que define

que *tasks* deve ser um conjunto de *TASK* e que o tamanho máximo permitido para o conjunto *tasks* é de dez elementos.

4.2.2 Especificação das operações da máquina

Nas operações da máquina é especificado o comportamento dinâmico do sistema. É através das operações que o estado da máquina é alterado, respeitando sempre as suas restrições. Mais especificamente, as condições declaradas no invariante da máquina devem ser sempre satisfeitas ao final da operação, levando assim a máquina a um estado válido.

O cabeçalho de uma operação é composto por um nome, uma lista de parâmetros de entrada e uma lista de parâmetros de saída [2], sendo os parâmetros de entrada e os parâmetros de saída argumentos opcionais. Um exemplo de uma operação com parâmetros de entrada e saída pode ser visto na figura 4.3. Nela, o nome da operação é *query_task*, o parâmetro de entrada é *task* e o parâmetro de saída *ans*.

$ans \leftarrow query\_task(task) =$
PRE $task \in TASK$
THEN
    IF $task \in tasks$
    THEN $ans := yes$
    ELSE $ans := no$
END

Figura 4.3: Operação que consulta se uma tarefa pertence a máquina *Kernel*.

A operação propriamente dita é formada por pré-condição e corpo da operação. Na pré-condição são colocadas as informações sobre todos os parâmetros de entrada e as condições que devem ser satisfeitas para que a operação seja executada com sucesso. Com isso, a pré-condição funciona como uma premissa que deve ser suprida para que a operação funcione corretamente. Por exemplo, na figura 4.4, tem-se a operação *task_add* da máquina *kernel* (figura 4.2). Para que essa operação funcione corretamente, ou seja, não leve a máquina para um estado inválido, as pré condições $task \in TASK$, $task \notin tasks$ e $\mathbf{card}(tasks) < 10$ devem ser obedecidas. Assim, para o seguro funcionamento dessas operação *task* deve ser do tipo *TASK* e não deve pertencer ao conjunto *tasks* e esse conjunto deve conter menos de 10 elementos.

No corpo da operação é especificado o seu comportamento. Neste, os parâmetros de saída devem ser obrigatoriamente valorados e os estados da máquina podem ser alterados ou consultados. Assim, para realizar, de um modo formal as atualizações de estado e definições de parâmetros de saída, a notação de máquina

[2] A notação de máquina abstrata permite que uma operação retorne mais de um valor.

$$
\begin{array}{l}
task\_add(task) = \\
\quad \textbf{PRE} \\
\quad\quad task \in TASK \wedge \\
\quad\quad task \notin tasks \wedge \\
\quad\quad \textbf{card}(tasks) < 10 \\
\quad \textbf{THEN} \\
\quad\quad tasks := tasks \cup \{task\} \\
\quad \textbf{END};
\end{array}
$$

Figura 4.4: Operação inclui uma nova tarefa na máquina *Kernel*

abstrata possui um conjunto de atribuições abstratas, denominadas substituições, as quais serão explicadas a seguir.

Formalmente, uma substituição é um transformador de predicados (ou de fórmulas). Elas funcionam da seguinte maneira: se P for um predicado, $[S]P$ é um predicado resultado da aplicação da substituição de S à P. A seguir, são demonstradas algumas das principais substituições da AMN e como são transformados os predicados utilizados por elas.

Substituição simples

A substituição simples é definida da seguinte forma:

$$x := E$$

Nela, x é uma variável de máquina ou parâmetro de saída, para o qual será atribuído o valor da expressão E. Mais precisamente, uma substituição simples é interpretada da seguinte maneira:

$$[x := E]P \Rightarrow P(x \setminus E)$$

Onde se tem que, no predicado P, a variável x deve ser substituída por E.

Substituição múltipla

A substituição múltipla é uma generalização da substituição simples. Ela permite que várias variáveis sejam atribuídas simultaneamente. Uma substituição múltipla utilizando duas variáveis tem a seguinte forma:

$$x, y := E, F$$

Na definição acima, às variáveis x e y são substituidas pelas expressões E e F, respectivamente. Assim, da mesma forma que a substituição simples, a substituição múltipla é definida da seguinte maneira:

$$[x := E, y := F]P \Rightarrow P[E, F \setminus x, y]$$

Na qual, no predicado P, as variáveis x e y são substituidas por E e F, respectivamente. Por exemplo, $[x, y := y+5, x+10]$ resulta em $y+5 < x+10$.

Substituição condicional

As substituições simples e múltipla permitem somente uma opção de especificação, onde uma atribuição é sempre feita de maneira uniforme, sem opções e sem levar em consideração os estados iniciais da operação. Entretanto, as linguagens de programação convencionais disponibilizam um tipo condicional de atribuição, na qual são permitidos caminhos diferentes escolhidos de acordo com expressões lógicas que utilizam os valores iniciais das variáveis do sistema e os parâmetros passados.

Como nas linguagens de programação, a notação de máquina abstrata também permite a construção de atribuições condicionais, as quais são feitas através da substituição condicional. Com isso, uma substituição condicional funciona da mesma forma que nas linguagens de programação. Nela, inicialmente uma expressão lógica é avaliada e, em seguida o caminho que a estrutura deve seguir, quais atribuições devem ser realizadas, é escolhido de acordo com o resultado da expressão. A forma como é especificada uma substituição condicional na ANM pode ser vista a seguir:

IF E **THEN** S **ELSE** T **END**

Na especificação acima, S e T são substituições quaisquer. Elas tem as suas aplicações condicionadas pela expressão lógica E, que pode conter variáveis da máquina e parâmetros de entrada. Com isso, caso E seja satisfeita a substituição S é realizada e, caso contrário, a substituição T é executada. Assim, uma substituição condicional pode ser interpretada da seguinte forma:

$$[\textbf{IF}\ E\ \textbf{THEN}\ S\ \textbf{ELSE}\ T\ \textbf{END}]P \quad = \quad (E \Longrightarrow [S]P) \wedge (\neg E \Longrightarrow [T]P)$$

.

Nessa interpretação, se E for verdadeiro a substituição S é aplicada ao predicado P. Caso contrário, a substituição T é aplicada ao predicado P.

Um exemplo simples da utilização dessa substituição pode ser visto na figura 4.3. Nela, a expressão $task \in TASK$ é primeiramente analisada para decidir qual substituição simples deve ser aplicada. Caso o resultado da expressão seja afirmativo $ans := yes$ é aplicada e, caso a expressão seja negativa, $ans := no$ é aplicada.

Substituição não determinística ANY

As substituições vistas até agora seguem uma metodologia determinística, ou seja, são substituições que possuem um comportamento previsível, levam a apenas um resultado final pré-determinado. Entretanto, as máquinas abstratas em B

são utilizadas para fazer especificações iniciais de sistemas ou componentes e, na maioria das vezes, no início de uma especificação, o comportamento do sistema não é totalmente conhecido. Assim, para especificar o não-determinismo inicial de uma especificação, a notação de máquina abstrata disponibiliza um tipo especial de substituição denominada de substituição não determinística.

Substituições não determinísticas são substituições que introduzem escolhas aleatórias no corpo da operação, levando-a a um conjunto de estados finais diferentes a cada execução. Em uma substituição não determinística, a especificação define apenas o conjunto sobre o qual deve ser feita a escolha, abstraindo assim informações de como tal escolha deve ser realizada. Em outras palavras, em uma substituição não determinística existe um conjunto de estados finais possíveis, que podem ser alcançados a cada execução da substituição.

Uma substituição não determinística definida na AMN é a substituição **ANY**. Essa substituição possui o seguinte formato:

$$\textbf{ANY} \quad x \quad \textbf{WHERE} \quad Q \quad \textbf{THEN} \quad T \quad \textbf{END}$$

Através da definição acima, percebe-se que a substituição **ANY** é formada por três elementos:

x é uma lista de variáveis que serão utilizadas no corpo T da substituição. Essa variáveis serão restringidas pelo predicado Q;

Q predicado que delimita o conjunto de valores para as variáveis x. Nessa parte, as variáveis x devem obrigatoriamente ser tipadas; e

T é uma substituição que utiliza-se das variáveis x para atualizar estados ou atribuir valores para os parâmetros de saída da operação.

Um exemplo da substituição **ANY** pode ser visto na operação da figura 4.5. Nela, uma tarefa é aleatoriamente adicionada na máquina *Kernel*. Para isso, primeiramente a variável *task* é criada para representar um valor aleatório. Em seguida, o tipo e a restrição sobre *task* são definidos, conjunto de valores possíveis de *task*. Por último, a variável *task* é adicionada ao conjunto *tasks*. Assim, um comportamento não determinístico é atribuído à operação, pois para cada execução da operação a variável *task* pode assumir um valor qualquer de um conjunto definido.

Uma definição para a substituição **ANY** seria:

$$[\textbf{ANY} \quad x \quad \textbf{WHERE} \quad Q \quad \textbf{THEN} \quad T \quad \textbf{END}]P \Rightarrow \forall x.(Q \Rightarrow [T]P)$$

Indicando que, para todo valor que for escolhido para o conjunto de variável x que satisfaça Q, a substituição T devem estabelecer o predicado P.

4.3 Obrigações de prova

Após a criação de uma máquina abstrata utilizando o método B, ela deve ser avaliada estaticamente para saber se a mesma é coerente e passível de implemen-

$$
\begin{array}{l}
random\_create = \\
\textbf{PRE} \\
\quad \textbf{card}(tasks) < 10 \\
\textbf{THEN} \\
\quad \textbf{ANY} \\
\quad\quad task \\
\quad \textbf{WHERE} \\
\quad\quad task \in TASK \wedge \\
\quad\quad task \notin tasks \\
\quad \textbf{THEN} \\
\quad\quad tasks := tasks \cup \{task\} \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figura 4.5: Operação que cria uma tarefa aleatéria na máquina *Kernel*.

tação. Para realizar tal avaliação, o método B dispõe de um conjunto de obrigações de prova, que são expressões lógicas geradas a partir de uma especificação em B.

Resumidamente, a análise estática de uma máquina abstrata, através das obrigações de prova, avalia primeiramente se a máquina possui estados válidos, ou seja, se pelo menos uma combinação dos estados é alcançada pela máquina. Caso a máquina possua estados válidos, é avaliado se estes são alcançados na inicialização da máquina e ao final de cada operação. Com isso, as principais obrigações de prova gerada em uma máquina abstrata são: consistência do invariante, obrigação de prova da inicialização e obrigação de prova das operações. A seguir, tem em maior detalhe cada uma dessas obrigações de prova e como elas são geradas.

4.3.1 Consistência do Invariante

Nessa obrigação de prova, é analisado se o invariante da máquina possui pelo menos uma combinação em que todas variáveis tenham valores válidos, ou seja, a máquina possui pelo menos um estado válido. Essa obrigação de prova é definida da seguinte maneira:

$$\exists v.I$$

Onde v indica o vetor de todos as variáveis da máquina e I representa o invariante da máquina. Com isso, a definição acima pode ser entendida como: deve existir pelo menos um valor para o vetor de variáveis v que satisfaça o invariante I.

Um exemplo da aplicação dessa obrigação de prova na máquina da figura 4.2 seria:

$$\exists tasks.(tasks \in \mathrm{P}(TASK) \wedge \mathbf{card}(tasks) \leq 10)$$

O que pode ser provado como verdadeiro instanciando *tasks* com $\emptyset$, por exemplo.

4.3.2 Obrigação de prova da inicialização

Outra obrigação de prova necessária na análise estática da máquina abstrata é a obrigação de prova da inicialização. Nela, é analisado se os estados iniciais da máquina satisfazem seu invariante. Isso significa verificar se os estados iniciais da máquina são estados válidos. Assim, essa obrigação de prova é definida da seguinte maneira:

$$[T]I$$

Nesta, $[T]$ indica as substituições realizadas na inicialização da máquina e I indica as restrições definidas no invariante. Com isso, a obrigação de prova da inicialização da máquina da figura 4.2 é:

$$[task := \emptyset](tasks \in \mathrm{P}(TASK) \wedge \mathbf{card}(tasks) \leq 10) \quad \Rightarrow \quad \emptyset \in \mathrm{P}(TASK) \wedge \mathbf{card}(\emptyset) \leq 10$$

O que pode ser facilmente provado como válido.

4.3.3 Obrigação de prova das operações

Na obrigação de prova das operações deve ser analisado se, quando satisfeita a sua pré-condição, a execução da operação, a partir de um estado válido, levará a máquina a um estado válido. Assim a definição dessa obrigação de prova pode ser vista da seguinte maneira:

$$I \wedge P \Rightarrow [S]I$$

Na definição acima, I representa o invariante da máquina, P representa a pré-condição da operação analisada e S indica as substituições realizadas no corpo da operação. Uma explicação mais precisa dessa definição seria: quando a máquina estiver em um estado válido e a pré-condição da operação for satisfeita, a execução da operação deve manter a máquina em um estado válido. Nota-se com isso, que esta obrigação de prova não é necessária nas operações que não alteram o estado da máquina, chamadas de operações de consulta, como a da figura 4.3, pois, nessas operações, apenas o valor do parâmetro de retorno é alterado.

Um exemplo de uma obrigação de prova da operação *task_add* da máquina da figura 4.4 pode ser visto a seguir:

$$(tasks \in \mathrm{P}(TASK) \wedge \mathbf{card}(tasks) \leq 10) \wedge$$
$$(task \in TASK \wedge task \notin tasks \wedge \mathbf{card}(tasks \leq 10)) \Rightarrow$$
$$([tasks := tasks \wedge \{task\}]((tasks \in \mathrm{P}(TASK) \cup \mathbf{card}(tasks) \leq 10))))$$

4.4 Refinamento

A linguagem abstrata demonstrada até agora é usada principalmente para criar uma modelagem funcional de sistemas e componentes. Nesta, o principal objetivo é descrever o comportamento do sistema sem se preocupar com detalhes de como tal comportamento será implementado ou de como os dados serão manipulados pelo computador. Entretanto, para realizar uma modelagem mais concreta e passível de implementação, é necessário que notações matemáticas abstratas utilizadas na modelagem do sistema, como conjuntos e substituições não determinística, sejam descritas de forma mais concreta, o que é possível através do refinamento. Além disso, o refinamento pode ser usado para construir um modelo abstrato de forma incremental com a adição sucessiva de formalizações dos requisitos[**?**].

Através da técnica de refinamento, o método B possibilita um desenvolvimento gradativo do sistema. Nele, um sistema é especificado em estágios que vão da modelagem abstrata até um nível algorítmico denominado de implementação. Entre esses níveis de abstração existem modelos intermediários chamados de refinamentos, que combina especificações abstratas com detalhes de implementação.

Mais precisamente, refinamentos são decisões de projeto, nas quais estruturas abstratas são detalhadas em um nível mais concreto (refinamento vertical), ou extenções da especificação para adicionar novos requisitos(refinamento horizontal). Com isso, um refinamento deve obrigatoriamente estar ligado a um modelo abstrato anterior e possuir seu comportamento delimitado pelo modelo a qual está relacionado. Para garantir que essa relação entre módulos seja feita de forma coerente, existem mecanismos de análise estática denominados obrigações de prova do refinamento, os quais seráo detalhados na seção 4.4.3.

A construção de um refinamento é muito parecida com a construção de uma máquina abstrata. Assim, como na máquina abstrata, é dividido em seções onde são especificadas as informações do sistema. Um refinamento possui basicamente as mesmas seções de uma máquina abstrata, a diferença está nas seções, **REFINEMENT** e **REFINES**, onde são colocados respectivamente o nome do refinamento e o módulo que será refinado.

Como ocorreu na seção de máquina abstrata 4.2, para um melhor entendimento, a especificação de um refinamento será dividida basicamente em duas partes principais: refinamento do estado da máquina abstrata e refinamento das operações da máquina abstrata. Em seguida, tem-se o delineamento dessas duas partes principais.

4.4.1 Refinamento do Estado

No refinamento de dados, como é reconhecido o refinamento do estado, tem-se o objetivo de especificar o estado de uma máquina em uma forma mais concreta, ou seja, mais próxima a utilizada pelo computador. Para isso, estruturas abstratas, como conjuntos e relações, são substituídas por estruturas de dados como vetores e sequências.

Como foi dito anteriormente, um refinamento necessita estar relacionado com um nível abstrato. No refinamento de dados, essa relação é feita através de um mecanismo denominado *relação de refinamento*. Assim, a *relação de refinamento*, nada mais é do que formulações lógicas que ligam o estado do refinamento ao estado do módulo refinado por ele.

Um exemplo de refinamento de dados pode ser visto na figura 4.6. Nela, é feito o refinamento da máquina *Kernel* (figura 4.2), a qual possui o estado *task* especificado como sendo um conjunto de tarefas. Entretanto, conjuntos são representações abstratas de dados. Assim, no refinamento *Kernel_r*, o estado *task* é refinado por *task_r*, uma sequência de tarefas ($tasks\_r \in \mathbf{seq}(TASK)$), estrutura mais concreta que um conjunto. A relação de refinamento entre os dois estados é feita através da igualdade $\mathbf{ran}(tasks\_r) = tasks$.

```
REFINEMENT   INVARIANT                  OPERATIONS
  Kernel_r     tasks_r ∈ seq(TASK) ∧    task_add(task) =
REFINES        ran(tasks_r) = tasks       BEGIN
  Kernel     INITIALISATION               tasks_r :=
VARIABLES      tasks_r := []                    task → tasks_r
  tasks_r                                 END
                                        END
```

Figura 4.6: Refinamento da maquina abstrata de *Kernel*.

4.4.2 Refinamento das operações

Após o refinamento do estado da máquina, é necessário especificar o refinamento das suas operações. Nesse processo, as operações abstratas são reescritas de forma mais concreta, podendo manipular o estado da máquina refinada e o novo estado criado no refinamento. Além disso, as operações refinadas devem possuir o mesmo comportamento das operações abstratas, garantindo assim uma coerência com a especificação inicial.

As operações de um refinamento devem possuir a mesma assinatura das operações do módulo relacionado a ele, ou seja, ter os mesmo nomes e parâmetros de entrada e saída. Entretanto, nas operações de um refinamento, não é necessária a declaração da pré condição (**PRE**), uma vez que essa foi definida em um nível mais abstrato e é suficiente para garantir que o tipo do parâmetro de entrada permaneça o mesmo.

Um exemplo do refinamento de uma operação pode ser visto na operação *task_add* do refinamento $Kernel_r$ (figura 4.6). Nela, percebe-se a ausência da pré-condição e que a assinatura da operação permanece a mesma. A parte alterada foi apenas o corpo da operação, que foi adaptada para trabalhar com o estado *taskR*. Assim, na operação da figura 4.4, a operações *task_add* adicionava um novo elemento a um conjunto de tarefas e, nessa nova operação, da figura 4.6, uma nova tarefa é adicionada no início de uma sequência de tarefa pela atribuição $tasks\_r := task \rightarrow tasks\_r$.

4.4.3 Obrigação de prova do refinamento

A análise estática, que confere se um refinamento é consistente com o nível abstrato acima dele, é feita através de obrigações de prova e pode ser dividida em duas partes, obrigação de prova da inicialização e obrigação de prova das operações. Entretanto, na obrigação de prova das operações, são possíveis ainda dois tratamentos diferentes, obrigações de prova para as operações sem parâmetros de retorno e a obrigação de prova para as operações com parâmetros de retorno. A seguir é demonstrado como é realizada cada uma dessas obrigações de prova do refinamento.

Obrigação de prova da inicialização

Em geral, a inicialização da máquina abstrata, nomeada de T, e a inicialização do refinamento, nomeada de *T1*, possuem um conjunto de execuções possíveis que levam a um conjunto de diferentes estados. Assim, em um refinamento, é necessário que cada execução de *T1* possua uma execução correspondente em T. Em outras palavras, todo estado encontrado em *T1* deve possuir, via *relação de refinamento*, denominada J, um estado gerado por T.

A *relação de refinamento* é um predicado entre variáveis abstratas e variáveis de refinamento. Com isso, T deve possuir pelo menos uma transição que satisfaça esse predicado, ou seja, nem todas as transições de T levará J a falsidade. Essa afirmativa pode ser traduzida na expressão abaixo:

$$\neg[T]\neg J$$

O predicado $\neg J$ indica que J é falso e o predicado $[T]\neg J$ representa que toda transformação de T levará J a um estado falso. Assim, a negação dessa afirmação $\neg[T]\neg J$ indica que existe uma transição de T que não levará J é falsidade, ou seja, nem todas transições de T levará J a falsidade.

Para encerrar a obrigação de prova da inicialização do refinamento, é necessário que para toda transformação de *T1* o predicado $\neg[T]\neg J$ seja estabelecido. Essa afirmação pode ser traduzida na expressão abaixo, a qual representa a obrigação de prova do refinamento.

$$[T1]\neg[T]\neg J$$

Um exemplo de uma obrigação de prova da inicialização do refinamento pode ser visto entre as máquina *Kernel* e a máquina *Kernel_r*. Nela, a inicialização dos estados das duas máquinas gera a seguinte obrigação de prova:

$$[tasks\_r := []]\neg[tasks := \emptyset]\neg(\mathbf{ran}(tasks\_r) = tasks)$$

Obrigação de prova da operação sem parâmetro de retorno

Geralmente, uma operação é definida como **PRE** *P* **THEN** *S* **END**, sendo o seu refinamento **PRE** *P1* **THEN** *S1* **END**, onde, na maioria da vezes, *P1* reduz-se a *TRUE*. Com isso, do mesmo modo que na inicialização, tem-se que as transições geradas por *S1* devem estar relacionadas com alguma transição de *S*, o que é definido pela expressão abaixo:

$$[S1]\neg[S]\neg J$$

Entretanto, diferente da inicialização, a execução de uma operação deve levar em consideração o estado da máquina anterior a sua execução. Assim, o estado da máquina abstrata junto com o estado do seu refinamento devem ser estados válidos. Uma relevante ligação entre esse estados é o invariante I e a sua relação de refinamento J. Além disso, para a correta execução da operação, a pré-condição da mesma deve ser estabelecida. Assim, levando em consideração que uma operação só pode ser executada corretamente quando a máquina estiver em um estado válido e quando a sua pré-condição for estabelecida, a obrigação de prova de uma operação é feita da seguinte forma:

$$I \wedge J \wedge P \Rightarrow [S1]\neg[S]\neg J$$

Por exemplo, a obrigação de prova do refinamento da operação *task_add* da máquina *Kernel* é :

$$\begin{array}{l}(tasks \in \mathrm{P}(TASK) \wedge \mathbf{card}(tasks) \leq 10)\wedge \\ \mathbf{ran}(tasks\_r) = tasks\wedge \\ task \in TASK\wedge \\ task \notin tasks\wedge \\ (\mathbf{card}(tasks) < 10) \Rightarrow [tasks := tasks \cup \{task\}] \\ \neg[tasks\_r := task \rightarrow tasks\_r] \\ \neg(tasks\_r) = tasks)\end{array}$$

Obrigação de prova da operação com parâmetro de saída

No refinamento de uma operação com saídas, cada saída do refinamento da operação deve estar ligada a uma saída da operação refinada, sendo necessário assim que a operação do refinamento tenha a mesma quantidade de parâmetros de saída da operação refinada. Além disso, renomeando out' como o conjunto de

parâmetros de saída do refinamento e deixando *out* como o conjunto dos parâmetros de saída da operação refinada, cada valor de out' deve possuir um correspondente em *out*. Em outra palavra, cada execução de *S1* deve encontrar uma execução *S*, na qual out' produzido por *S1* seja igual ao *out* produzido por *S*.

Além da ligação entre os parâmetros de saída, no refinamento de uma operação com retorno, deve-se obedecer todas as restrições impostas no refinamento das operações sem parâmetro de saída, ficando obrigação de prova para as operações com retorno da seguinte forma:

$$I \wedge J \wedge P \Rightarrow S1[out'/out]\neg S\neg(J \wedge out' = out)$$

Nela, $S1[out'/out]$ significa que, nas atribuições de *S1*, cada ocorrência de *out* deve ser substituída por out' e, antes dessa substituição, a máquina deve possuir um estado válido e sua pré-condição deve ser alcançada, $I \wedge J \wedge P$. As demais verificações são similares à obrigações de prova das operações sem parâmetros de saída.

Através desse capítulo pode-se perceber como é feita uma especificação em B e principalmente a sua organização em módulos bem definidos. Além disso, como o mecanismo de refinamento é possível chegar-se em níveis de especificação próximo às linguagens de programação imperativas, nos quais o modelo criado é passível de transformação para uma linguagem executável.

Entretanto, devido ao seu rigor matemático, a linguagem possui limitações que dificutam o trabalho do especificador. Entre elas está o fato de que uma operação não poder ser reaproveitada na mesma máquina. Por exemplo, a operação *task_delete* da máquina *Kernel*(figura 4.2) não poderia ser reaproveitada na operação na mesma máquina se essa nova operação necessitasse excluir uma tarefa, impedindo assim a reutilização de código.

Capítulo 5

Escopo da especificação

Nos trabalhos discutidos no capítulo 2, uma das principais técnicas utilizadas foi limitar o escopo da especificação para tornar viável o desenvolvimento do projeto. Seguindo o mesmo raciocínio, na modelagem do FreeRTOS, apenas algumas das suas abstrações de hardware, junto com suas características, foram especificadas nessa dissertação. Com isso, a parte do sistema a ser tratada nesse trabalho será demonstrada a seguir.

5.1 Escopo da especificação

Durante o estudo da documentação do FreeRTOS percebeu-se que algumas abstrações de hardware são mais importantes e mais utilizadas em aplicações criadas a partir do FreeRTOS. Devido a isso, decidiu-se que as entidades a serem tratadas nesse trabalho serão: tarefas, fila de mensagens, semáforos e mutex.

As tarefas foram escolhidas porque são as unidades básicas de execução do sistema, consequentemente, requisito indispensável para o modelo. Do mesmo modo, as filas de mensagens são as estruturas primárias de comunicação entre as tarefas, pois é dela que derivam os semáforos e mutex, sendo esses estruturas básicas de sicronização entre tarefas.

Após a decidir quais abstrações serão tratadas na especificação, foram selecionadas as características de cada entidade a serem modeladas. Como resultado dessa seleção, criou-se as listadas abaixo, nas quais cada característica está organizada de acordo com as entidades que pertencem.

Ressalta-se, que as características aqui comentadas estão na sua forma mais genérica. Assim, cada requisito abaixo pode ser composto por uma ou mais características específicas, que serão comentadas no decorrer da explicação sobre o modelo criado.

Tarefas

Para a entidade tarefa as características escolhidas nessa especificação foram:

- Armazenar todas as tarefas gerenciadas pelo FreeRTOS;

- Estados de uma tarefa;
- Prioridade de uma tarefa;e
- Política de escalonamento (escolha da próxima tarefa a entrar em execução).

Ainda, como funcionalidade que foi colocada junto com o conjunto de tarefas, tem-se o tempo de execução do escalonador, ou seja, o periodo no qual o sistema está em funcionamento.

Fila de Mensagens

Na fila de mensagens este trabalho preocupou-se em:

- Armazenar todas as filas de mensagens gerênciadas pelo FreeRTOS;
- Gerenciar as mensagens da fila;
- Controlar os estados de uma fila (fila cheia ou fila vazia);e
- Gerenciar as tarefas bloqueadas por escrita ou leitura.

Semáforo

A entidade semáforo abrange tantos os semáforos binários, com os semáforos com contador. Os caracteres dessas abstrações tratados foram:

- Armazenar os semáforos manuseados pelo FreeRTOS;
- Gerenciar as tarefas que utilizam o semáforo; e
- Gerenciar os estados dos semáforos (disponível e ocupado).

Mutex

Para o mutex, as principais características tratadas foram:

- Armazenar os mutexes pelo FreeRTOS;
- Gerenciar as tarefas que utilizam mutex;
- Gerenciar os estados dos mutex; e
- Controlar o mecanismo de herança de prioridade.

Funcionalidades da API

Vale lembrar que, nesse trabalho, foram tratadas principalmente as funcionalidades da API do FreeRTOS relacionada a esses elementos. Entretanto, na especificação dessas funcionalidades preocupou-se apenas com as características listadas nessa seção. Por exemplo, a funcionalidade *xTaskCreate*, responsável por criar uma nova tarefa, aloca um espaço na memória para a pilha da tarefa, coloca a prioridade da tarefa, escolhe o estado da tarefa, associa um nome a tarefa e realiza outras particularidades do sistema para a criação de uma tarefa. Na especificação, essa funcionalidade preocupa-se apenas com o estado da tarefa e

a sua prioridade e outros aspectos podem ser tratados futuramente, através de refinamentos.

Nesse contexto, as funcionalidades da API do FreeRTOS tratadas nesse trabalho foram:

xTaskCreate,
uxTaskPriorityGet,
vTaskResume,
uxTaskGetNumberOfTasks,
vTaskDelayUntil,
vTaskStartScheduler,
vTaskSuspendAll,
vTaskPrioritySet,
xQueueSendToFront,
xQueuePeek,
xQueueSend,
xQueueSendToFront,
xQueuePeek,
xSemaphoreTake,
vSemaphoreCreateCounting,
xSemaphoreTake,
vTaskDelete,
vTaskSuspend,
xTaskGetCurrentTaskHandle,
vTaskDelay,
xTaskGetTickCount,
vTaskEndScheduler,
xTaskResumeAll,
xQueueSendToBack,
xQueueReceive,
xQueueCreate,
xQueueSendToBack,
xQueueReceive,
xSemaphoreGive,
vSemaphoreCreateBinary,
xSemaphoreCreateMutex,
xSemaphoreGive

Capítulo 6

Modelagem Inicial

Após a definição do escopo do trabalho, foi criada uma modelagem funcional que possuía um núcleo das funcionalidades listadas no capítulo anterior. Essa especificação utilizou-se principalmente das técnicas de modularização e desenvolvimento incremental do método B, as quais são explicadas a seguir:

- Parte dos requisitos funcionais do sistema podem ser abstraídos em sua modelagem inicial. Assim, tais requisitos são tratados posteriormente através de refinamentos horizontais ou extensões da especificação, o que proporciona uma especificação incremental do sistema. O planejamento incremental dessa especificação será explicado nas seções 6.0.1 e 6.0.2.
- Quando os requisitos do sistema não apresentarem dependências entre si, eles podem ser especificados em diferentes módulos. Esses módulos comunicam entre si utilizando os mecanismos de composição proporcionados pelo método B (visão, inclusão, etc). A divisão da especificação em módulos será detalhada na seção 6.1.

Por serem as entidades base do FreeRTOS, nesse modelo funcional, foram tratadas somente as abstrações tarefa e fila de mensagens. Essas foram especificadas conforma a explicação abaixo:

6.0.1 Tarefa

Na entidade tarefa apenas a característica de estado de uma tarefa foi formalizada inicialmente. Através dessa formalização, propriedades importantes do sistema puderam ser especificadas. São essas:

1. Todas as tarefas manipuladas pelo sistema devem ser armazenadas.
2. A troca de estado das tarefas devem ocorrer como demonstra a figura 3.2.
3. Uma tarefa deve possuir somente um estado em determinado momento.
4. Quando o escalonador estiver ativo, todas as tarefas devem estar no estado pronta.
5. Quando o escalonador estiver ativo, a tarefa ociosa deve estar no estado pronta ou em execução.

6. Deve existir somente uma tarefa em execução.

A especificação da entidade tarefa foi planejada de forma incremental. Previamente, foram criados os estados necessários para modelar tal entidade no nível de abstração sugerido. Após isso, algumas das funcionalidades do sistema, relacionadas a essa entidade, foram formalizadas a cada etapa de criação do modelo. Ao final, as seguintes funcionalidades foram abrangidas de forma abstrata por essa especificação inicial:

- Criação de tarefas: *xTaskHandle*, *xTaskCreate*, *vTaskDelete*.
- Controle de tarefas: *vTaskDelay*, *vTaskDelayUntil*, *uxTaskPriorityGet*, *vTaskPrioritySet*, *vTaskSuspend*, *vTaskResume*.
- Utilitários de tarefas:*xTaskGetCurrentTaskHandle*, *uxTaskGetNumberOfTasks*, *xTaskGetTickCount*, *xTaskGetSchedulerState*.
- Controle do escalonador: *vTaskStartScheduler*, *vTaskEndScheduler*, *vTaskSuspendAll*, *xTaskResumeAll*.

6.0.2 Fila de mensagens

Para a entidade fila de mensagens, a principal característica especificada foi a quantidade de mensagens que esta pode armazenar. Como ocorreu com a entidade Tarefa, a formalização desta, junto com suas funcionalidades, foi distribuída entre as várias etapas de desenvolvimento da modelagem inicial. Com essa especificação, as seguintes propriedades do sistemas foram tratadas pela modelagem:

1. Todas as filas de mensagens manipuladas pelo sistema devem ser armazenadas.
2. Gerenciar os itens armazenados pela fila.
3. Gerenciar as tarefas bloqueadas por leitura na fila.
4. Gerenciar as tarefas bloqueadas por escrita na fila.

As funcionalidades relacionadas à fila de mensagens tratadas nessa especificação foram: *xQueueCreate*, *vQueueDelete*, *xQueueSend*, *xQueueSendToBack*, *xQueueSendToFront*, *xQueueReceive*, *xQueuePeek*.

6.1 A modelagem funcional

A modelagem discutida nas seções anteriores foi construída e verifica utilizando a ferramenta AtelierB 4.0[**?**]. Tal especificação foi estruturada através de sete módulos, os quais serão explicados a seguir:

Módulo Config: Nesse módulo foram tratadas as configurações do comportamento do sistema. Por exemplo, a prioridade máxima que uma tarefa pode possuir é um parâmetro de configuração desse módulo.

Módulo Types: Esse módulo é responsável por definir os tipos utilizados na especificação do FreeRTOS. Um exemplo de um tipo especificado nesse módulo é a prioridade de uma tarefa, que só pode assumir valores de um subconjunto finito dos naturais.

Módulo Task: Nesse módulo são definidos os estados e operações responsáveis por formalizar a entidade tarefa.

módulo Queue: Esse módulo é similar ao módulo Task, só que nele a entidade especificada é a fila de mensagens.

Módulo Scheduler: Esse módulo simplesmente mantém e manipula o estado do escalonador.

Módulo FreeRTOSBasic: Esse módulo funciona como uma camada abstrata entre o módulo *FreeRTOS* e os demais módulos básicos. Nele, as operações implementadas pelos módulos *Task*, *Queue* e *Scheduler* são agrupadas em funções mais abstratas, que servem como base para especificar as funcionalidades do *FreeRTOS*.

Módulo FreeRTOS: Finalmente, no módulo *FreeRTOS* são especificadas as funcionalidades das bibliotecas do FreeRTOS.

A organização desses módulos pode ser vista na figura 6.1. Nela, resumidamente os módulos inferiores servem de base para a especificação dos módulos superiores, restando os módulos *Config* e *Types*, que servem de apoio para toda a especificação. Nas seções seguintes tem-se em detalhe como foi o desenvolvimento dessa especificação inicial.

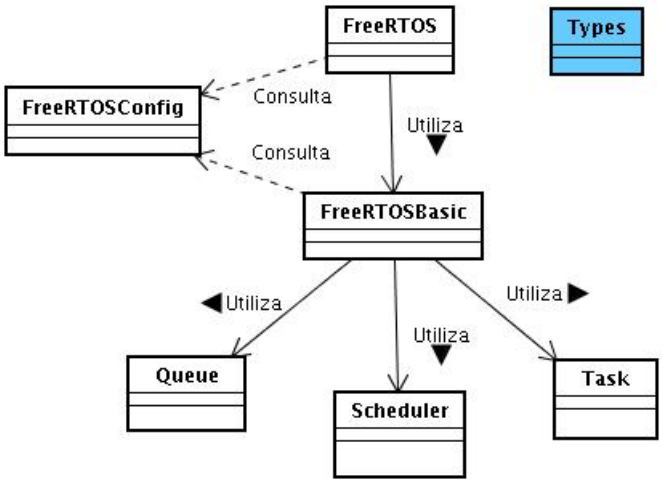


Figura 6.1: Esbouço da arquitetura da especificação.

6.1.1 Tarefa

Para representar o elemento tarefa existem diversas abordagens, funções de mapeamento, sequências e conjuntos. A abordagem escolhida nesse trabalho ini-

cial foi a representação através de conjuntos, o que facilita a verificação da especificação pelo provador de teoremas. Assim, a modelagem da entidade tarefa e suas propriedades foram feitas de acordo com a figura 6.2. Nela, a variável *active* informa quando o sistema está ativo. A variável *tasks* armazena todas as tarefas criadas no sistema (propriedade 1 da seção 6.0.1). Em seguida, as variáveis *ready*, *blocked*, *running* e *suspended* armazenam as tarefas dos estados em execução, pronta, bloqueada e suspensa, respectivamente. Por fim, a variável *idle* representa a tarefa ociosa.

Um exemplo do funcionamento da especificação da figura 6.2 seria o seguinte. Enquanto o sistema estiver inativo, o valor da variável *active* deve ser falso. As novas tarefas criadas no sistema serão armazenadas na variável *tasks* e na variável de seu respectivo estado, sendo a tarefa ociosa armazenada em *idle*. Por fim, ao iniciar-se a execução do sistema, a variável *active* receberá o valor verdadeiro.

MACHINE
Task
VARIABLES
$active, tasks, blocked, running, ready, suspended, idle$
INVARIANT
$active \in \mathbf{BOOL} \wedge tasks \in \mathbf{FIN}(TASK) \wedge running \in TASK \wedge idle \in TASK$
$\wedge blocked \in \mathbf{FIN}(TASK) \wedge ready \in \mathbf{FIN}(TASK) \wedge suspended \in \mathbf{FIN}(TASK)$

Figura 6.2: Especificação do estado do módulo *Task*.

Estendendo o módulo *Task*, tem-se, na figura 6.3, a continuação do invariante da máquina. Nele, a maioria das propriedades listadas na seção 6.0.1 foram tratadas. A propriedade 3 foi modelada com as seguintes asserções:

- $tasks = \{running\} \cup suspended \cup blocked \cup ready$;
- $ready \cap blocked = \emptyset \wedge blocked \cap suspended = \emptyset \wedge suspended \cap ready = \emptyset$; e
- $running \notin (blocked \cup ready \cup suspended)$.

A primeira informa que toda tarefa deve possuir um estado e as demais garantem que nenhuma tarefa deve ter mais de um estado ao mesmo tempo. Por fim, as propriedades 4 e 5 são tratadas em $(active = FALSE \Rightarrow tasks = ready)$ e $(active = TRUE \Rightarrow (idle = running \vee idle \in ready))$, respectivamente.

Continuando com a especificação do módulo *Task*, foram criadas as operações básicas para manipular as variáveis de estado da máquina. Essas operações serviram como base para a modelagem das funcionalidades do FreeRTOS relacionadas à entidade Tarefa. Ao total, doze operações elementares foram desenvolvidas, das quais quatro delas são apresentadas pelas figuras 6.4, 6.5, 6.7 e 6.6. As demais operações e suas obrigações de prova podem ser consultadas no repositório do projeto[1].

[1]Página do repositório do projeto: http://code.google.com/p/freertosb/

$$
\begin{array}{l}
\ldots \\
\quad blocked \subseteq tasks \wedge ready \subseteq tasks \wedge suspended \subseteq tasks \wedge \\
\quad ready \cap blocked = \emptyset \wedge blocked \cap suspended = \emptyset \wedge suspended \cap ready = \emptyset \wedge \\
\quad (active = FALSE \Rightarrow tasks = ready) \wedge \\
\quad (active = TRUE \Rightarrow \ (idle = running \vee idle \in ready) \wedge \\
\qquad\qquad\qquad\qquad running \notin (blocked \cup ready \cup suspended) \wedge \\
\qquad\qquad\qquad\qquad tasks = \{running\} \cup suspended \cup blocked \cup ready) \\
\ldots
\end{array}
$$

Figura 6.3: Continuação do invariante da máquina *Task*.

$$
\begin{array}{l}
result \longleftarrow t\_create(priority) = \\
\textbf{PRE} \\
\quad priority \in PRIORITY \wedge \\
\quad active = FALSE \\
\textbf{THEN} \\
\quad \textbf{ANY}\ task \quad \textbf{WHERE} \\
\qquad task \in TASK \wedge task \notin tasks \\
\textbf{THEN} \\
\quad tasks := \{task\} \cup tasks \parallel \\
\quad ready := \{task\} \cup ready \parallel \\
\quad result := task \\
\quad \textbf{END} \\
\textbf{END};
\end{array}
$$

Figura 6.4: Especificação da operação *t_create*.

Na figura 6.4, tem-se a operação *t_create*, que é responsável por criar um nova tarefa. Essa operação pode ser utilizada somente quando o escalonador não estiver acionado, como indica a sua pré-condição $active = FALSE$. Seu parâmetro de entrada, *priority*, informa a prioridade da tarefa que será criada. Entretanto, esse parâmetro só será utilizado no refinamento da operação. Para criar uma nova tarefa, representada por *task*, é usada a substituição **ANY** . Em seguida, a nova tarefa é adicionada no conjunto de tarefas manuseadas pelo FreeRTOS ($tasks := \{task\} \cup tasks$), colocada no estado pronta ($ready := \{task\} \cup ready$) e indicada como parâmetro de retorno ($result := task$).

$$
\begin{array}{l}
t\_startScheduler = \\
\textbf{PRE} \\
\quad active = FALSE \\
\textbf{THEN} \\
\quad active := TRUE \parallel \\
\quad blocked, suspended := \emptyset, \emptyset \parallel \\
\quad \textbf{ANY}\ idle\_task\ \textbf{WHERE} \\
\qquad idle\_task \in TASK \wedge \\
\qquad idle\_task \notin tasks \\
\quad \textbf{THEN} \\
\qquad tasks := \{idle\_task\} \cup tasks \parallel \\
\qquad idle := idle\_task \parallel \\
\qquad \textbf{ANY}\ task\ \textbf{WHERE} \\
\qquad\quad task \in ready \cup \{idle\_task\} \\
\qquad \textbf{THEN} \\
\qquad\quad running := task \parallel \\
\qquad\quad ready := (ready \cup \{idle\_task\}) - \{task\} \\
\qquad \textbf{END} \\
\quad \textbf{END} \\
\textbf{END};
\end{array}
$$

Figura 6.5: Especificação da operação *t_startScheduler*.

A segunda operação a ser explicada é a *t_startScheduler*, figura 6.5, responsável por iniciar o funcionamento do sistema. Nessa operação, a tarefa ociosa, representada por *idle_task*, é criada na substituição **ANY** e, em seguida, armazenada no

conjunto de tarefas do sistema e atribuida a variável responsável por especificar a tarefa ociosa. Após isso, na segunda substituição **ANY**, uma tarefa, representada por *task* é escolhida entre o conjunto de tarefas prontas união com a tarefa ociosa ($task \in ready \cup \{idle\_task\}$). Essa tarefa é colocada no estado em execução e retirada do estado pronta, para a iniciação do sistema.

Com as operações demonstradas anteriormente, percebe-se que a propriedade 2 da seção 6.0.1 é especificada no decorrer das operações da máquina *Task*. Um exemplo mais claro disso ocorre nas operações das figuras 6.6 e 6.7 responsáveis por colocar uma tarefa no estado bloqueada e retornar uma tarefa do estado suspensa, respectivamente.

```
t_delayTask(ticks) =
PRE
  active = TRUE ∧
  running ≠ idle ∧
  ticks ∈ TICK ∧
  ticks ≠ 0
THEN
  blocked := blocked ∪ {running} ||
  ANY
    task
  WHERE
    task ∈ TASK ∧
    task ∈ ready
  THEN
    ready := ready − {task} ||
    running := task
  END
END;
```

Figura 6.6: Especificação da operação *t_delayTask*.

Na operação *t_delayTask* da figura 6.6, é informado no parâmentro *ticks* o tempo no qual a tarefa deve permanecer no estado bloqueada, sendo este definido como $ticks \in TICK$. Em seguida, a tarefa em execução é colocada no estado bloqueado pela asserção $blocked := blocked \cup \{running\}$. Por fim, uma nova tarefa de estado pronta, representada por *task*, é escolhida para entrar em execução, utilizando a substituição **ANY**. Consequentemente, essa tarefa é retirada do estado pronta e colocada no estado em execução através da das substituições $ready := ready - \{task\}$ e $running := task$.

```
t_resume(atask) =
PRE
  active = FALSE ∧
  active = TRUE ∧
  atask ∈ TASK ∧
  atask ∈ suspended
THEN
  CHOICE
    ready := ready ∪ {running} ||
    running := atask
  OR
    ready := ready ∪ {atask}
  END ||
  suspended := suspended − {atask}
END;
```

Figura 6.7: Especificação da operação *t_resume*.

A operação responsável por retornar uma tarefa do estado suspensa, figura 6.7, tem como parâmentro de entrada a tarefa *atask*, que deve estar no estado sus-

pensa ($atask \in suspended$). Em seguida, essa tarefa é retirada do conjunto de tarefas suspensas e colocada no conjunto de tarefas prontas, respectivamente, pelas seguintes asserções: $suspended := suspended - \{atask\}$ e $ready := ready \cup \{atask\}$.

Especificação das funcionalidades da API

Após criação das operações básicas para manipulação das tarefas, a especificação das funcionalidades da API relacionadas a esta entidade tornou-se mais simples. Isso porque, para modelar uma funcionalidade especifica dessa abstração, basta apenas utilizar a operação básica responsável pelo requisito da funcionalidade.

Um exemplo da especificação de uma funcionalidade pode ser visto na figura 6.8. Nela, a funcionalidade *xTaskCreate*, responsável por criar uma nova tarefa, é demonstrada. O seu comportamento pode ocorrer de duas formas: ou uma nova tarefa do sistema é criada e passada como retorno da função; ou nenhuma tarefa é criada e uma mensagem de erro é retornada. Com isso, para criar uma nova tarefa, essa funcionalidade apenas utiliza-se da função *t_create* demonstrada na figura 6.4.

```
result, handle ⟵
        xTaskCreate( code, name,
                     stackSize, params,
                     priority) =
PRE
  code ∈ TASK_CODE∧
  name ∈ NAME∧
  stackSize ∈ NATURAL∧
  params ⊂ PARAMETER∧
  priority ∈ PRIORITY∧
  scheduler = NOT_STARTED∧
THEN
  CHOICE
    handle ←
        t_create(priority) ||
    result := pdPASS
  OR
    result := errMEMORY ||
    handle :∈ TASK
  END
END
```

Figura 6.8: Especificação da operação *xTaskCreate*.

Outro exemplo mais simples pode ser visto na especificação da funcionalidade *vTaskResume* na figura 6.9. Essa funcionalidade é reponsável por colocar uma tarefa suspensa no estado pronta. Para isso, ela utiliza-se da operação *t_resume* da máquina *Task*, demonstada na figura 6.7.

6.1.2 Fila de mensagens

A especificação da parte responsável pela abstração fila de mensagens seguiu o mesmo raciocínio da entidade tarefa. A abordagem escolhida para sua representação foi os conjuntos e uma máquina, denominada *Queue*, ficou responsável por tratar das suas características.

vTaskResume(*pxTaskToResume*) =
PRE
scheduler = *taskSCHEDULER_RUNNING*∧
INCLUDE_vTaskSuspend = 1∧
suspended ≠ ∅∧
pxTaskToResume ∈ *TASK*∧
pxTaskToResume ∈ *suspended*
THEN
t_resume(*pxTaskToResume*)
END;

Figura 6.9: Especificação da operação *vTaskResume*.

O estado da máquina *Queue* pode ser visto na figura 6.10. Nela, foram criadas as variáveis *queues*, *items*, *receiving*, e *sending*. A variável *queues* é responsável por armazenar todas as filas de mensagens criadas pelo sistema (propriedade 1 da seção 6.0.2). A variável *items* relaciona uma fila de mensagens a um conjunto de itens (propriedade 2 da seção 6.0.2). Por último, as variáveis *receiving* e *sending* relacionam, respectivamente, uma fila de mensagens a um conjunto de tarefas bloqueadas aguardando a chegada de uma mensagem na fila (propriedade 3 da seção 6.0.2) e um conjunto de tarefas bloqueadas aguardando para enviar uma mensagem para a fila (propriedade 4 da seção 6.0.2).

MACHINE	**INVARIANT**
Queue	$queues \in \mathrm{P}(QUEUE) \wedge$
VARIABLES	$items \in QUEUE \pfun \mathrm{P}(ITEM) \wedge \mathrm{dom}(items) = queues \wedge$
queues,	$receiving \in QUEUE \pfun \mathrm{P}(TASK) \wedge \mathrm{dom}(receiving) = queues \wedge$
items,	$sending \in QUEUE \pfun \mathrm{P}(TASK) \wedge \mathrm{dom}(sending) = queues$
receiving,	
sending	

Figura 6.10: Estado da máquina *Queue*

Na máquina *Queue*, também foram criadas operações básicas para manipular os estados anteriormente descritos. Ao total, seis operações foram desenvolvidas, entre elas as operações *xQueueCreate*, *sendItem*, e *insertTaskWaitingToRecived*, as quais serão explicadas a seguir.

A operação responsável por criar um nova fila de mensagens é a *xQueueCreate*, figura 6.11. Nela, os parâmetros de entrada *uxQueueLength* e *uxItemSize*, representam os tamanho da fila e o tamanho do item da fila, respectivamente. Em seguida, a nova fila de mensagens é escolhida em **ANY** pela variável *queue*. Para essa nova fila são atribuidos um conjunto vazio de itens ($queue\_items := queue\_items \cup \{queue \mapsto \emptyset\}$), um conjunto vazio de tarefas bloqueadas por leitura ($queue\_receiving := queue\_receiving \cup \{queue \mapsto \emptyset\}$) e um conjunto vazio de tarefas bloqueadas por escrita ($queue\_sending := queue\_sending \cup \{queue \mapsto \emptyset\}$). Por último, a nova fila criada é retornada pela a operação ($xQueueHandle := queue$).

$xQueueHandle \longleftarrow xQueueCreate(uxQueueLength, uxItemSize) =$
PRE
  $uxQueueLength \in \mathrm{N} \wedge$
  $uxItemSize \in \mathrm{N}$
THEN
  ANY
    $queue$
  WHERE
    $queue \in QUEUE \wedge$
    $queue \notin queues$
  THEN
    $queues := queues \cup \{queue\} \parallel$
    $queue\_items := queue\_items \cup \{queue \mapsto \emptyset\} \parallel$
    $queue\_receiving := queue\_receiving \cup \{queue \mapsto \emptyset\} \parallel$
    $queue\_sending := queue\_sending \cup \{queue \mapsto \emptyset\} \parallel$
    $xQueueHandle := queue$
  END
END;

Figura 6.11: Operação *xQueueCreate*.

$sendItem(queue, item, task, pos) =$
PRE
  $queue \in queues \wedge$
  $item \in ITEM \wedge$
  $task \in TASK \wedge$
  $pos \in COPY\_POSITION \wedge$
  $task \in receiving(queue)$
THEN
  $items(queue) :=$
    $items(queue) \cup \{item\} \parallel$
  $receiving(queue) :=$
    $receiving(queue) - \{task\}$
END

Figura 6.12: Especificação da função *sendIten*.

Na figura 6.12, tem-se a operação responsável por adicionar um item em uma fila de mensagens. Nela, os parâmetros de entrada *queue*, *item* e *task* representam a fila que será manipulada, o item que será adicionado à fila e a tarefa que será retirada do conjunto de tarefas bloqueadas aguardando a chegada de uma mensagem na fila. Por fim, devido aos conjuntos não possuírem posições, o parâmetro *pos*, que representa a posição de entrada do item na fila (final ou começo), não é utilizado.

$insertTaskWaitingToRecived(pxQueue, pxTask) =$
PRE
  $pxQueue \in queues \wedge$
  $pxTask \in TASK$
THEN
  $queue\_receiving(pxQueue) := queue\_receiving(pxQueue) \cup \{pxTask\}$
END;

Figura 6.13: Especificação da operação *insertTaskWaitingToRecived*.

Por último, a operação *insertTaskWaitingToRecived* da figura 6.13 adiciona uma tarefa no conjunto de tarefas bloqueadas por leitura em uma fila. Assim, a operação recebe a tarfa que será adicionada, *pxTask*, e a fila na qual a tarefa será adicionada ao conjunto, *pxTask*. Em seguida, a tarefa informada é adicionada ao conjunto de tarefas bloqueadas por leitura ($queue\_receiving(pxQueue) := queue\_receiving(pxQueue) \cup \{pxTask\}$).

A máquina FreeRTOSBasic

Prosseguindo na especificação, percebeu-se que as funcionalidades relacionadas à entidade fila de mensagens possuem um comportamento muito parecido. Além disso, para a especificação dessas funcionalidades, é necessário ter uma ligação com a entidade tarefa e fila de mensagens, pois uma fila é capaz de bloquear e desbloquear uma tarefa.

Nesse contexto, para especificar os comportamentos comuns das funcionalidades da fila de mensagens e fazer a ligação dessas funcionalidades com a entidade tarefa, foram criadas, na máquina *FreeRTOSBasic*, as seguinte funções intermediárias: *xQueueGenericSend* e *xQueueGenericReceive*. Essas funções são responsáveis respectivamente por: enviar um elemento para fila ou bloquear a tarefa remetente se a fila estiver cheia; receber um elemento da fila ou bloquear a tarefa receptora se a fila estiver vazia. A seguir, tem-se a especificação da operação *xQueueGenericSend*, podendo o leitor interessado verificar a especificação da operação *xQueueGenericReceived* no repositório do projeto.

```
res ← xQueueGenericSend(q, i, wait, pos) =
  PRE
    q ∈ queues ∧ i ∈ ITEM ∧ wait ∈ TICK ∧
    pos ∈ COPY_POSITION ∧
    active = TRUE ∧ running ≠ idle
  THEN
    CHOICE
      IF wait > 0 THEN
        q_insertTaskWaitingToSend(q, running) ||
        t_delayTask(wait) ||
        res := pdTRUE
      ELSE
        res := errQUEUE_FULL
      END
    OR
      ANY t WHERE
        t ∈ TASK ∧
        t ∈ blocked ∧
        t ∈ receiving(q)
      THEN
        q_sendItem(q, i, t, pos) ||
        t_unblock(t) ||
        res := pdPASS
      END
    END
  END
```

Figura 6.14: Especificação da função *xQueueGenericSend*.

Na função *xQueueGenericSend*, apresentada na figura 6.14, tem-se como parâmetros de entrada: a fila onde será enviada a mensagem, q ; o item que será enviado para a fila, i ; a quantidade de tempo que a tarefa remetente poderá ficar esperando pela fila, *wait*; e a posição que o item será colocado na fila, *pos*. Essa função possui dois comportamentos possíveis. No primeiro, representando uma fila cheia, ela bloqueia a tarefa remetente, através da operação *t_delayTask*,

e, com o método *q_insertTaskWaitingToSend*, coloca-a no conjunto de tarefas que esperam para enviar um item para fila. No segundo, o item é enviado para a fila, através do método *q_sendItem*, e uma tarefa que aguarda por um item da fila é desbloqueada com a operação *t_unblock*.

$$
\begin{array}{l}
res \longleftarrow xQueueSend(q,i,w) = \\
\textbf{PRE} \\
\quad q \in queues \wedge i \in ITEM \wedge w \in TICK \wedge \\
\quad active = TRUE \wedge running \neq idle \\
\textbf{THEN} \\
\quad res \leftarrow xQueueGenericSend(q,i,w,queueSEND\_TO\_BACK) \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Figura 6.15: Especificação da função *xQueueSend*.

Finalmente, para finalizar essa especificação inicial, a funcionalidade de enviar um item para a fila de mensagens, modelada na máquina *FreeRTOS*, é apresentada pela figura 6.15. Nela, basicamente, ocorre uma chamada à função *xQueueGenericSend*, da figura 6.14.

6.2 Refinando a especificação inicial

Para tratar dos requisitos do FreeRTOS ligados à prioridade de uma tarefa, é necessário acrescentar tal característica à modelagem inicial. Esse acréscimo está relacionado principalmente à máquina *Task* e pode ser feito de duas formas: escrevendo uma nova versão para essa máquina ou refinando-a. Aqui, a última forma foi escolhida como a mais adequada.

O refinamento da máquina *Task* é feito pelo módulo *Task_r*, cujas variáveis e invariantes são exibidos na figura 6.16. Nela, o tipo *PRIORITY* representa a prioridade que uma tarefa pode assumir e a variável *prio* representa a ligação de uma tarefa a sua prioridade. Com esse refinamento, foi possível, através do seu invariante, especificar o seguinte requisito: uma vez inicializado o escalonador, a tarefa em execução deve ter prioridade maior ou igual a das tarefas de estado pronta ($\forall t.(t \in ready \Rightarrow prio(t) \leq prio(running))$).

Feita as alterações para tratar a prioridade de uma tarefa, é necessário agora refinar as operações da máquina *Task*. No entanto, para facilitar e modularizar o refinamento dessas operações, foi criada previamente a função auxiliar *scheduler_p* no refinamento *Task_r*, que recebe como entradas um conjunto de tarefas e uma função que mapeia essas tarefas as suas prioridades. Ao final, essa função auxiliar retorna um conjunto de tarefas de maior prioridade. A especificação de *scheduler_p* pode ser vista na figura 6.17.

Finalmente, os refinamentos das operações previamente demonstradas, *t_create* (figura 6.4) e *t_startScheduler*(figura 6.5), são exibidos, respectivamente, pelas figuras 6.18 e 6.19.

REFINEMENT
Task_r
CONSTANTS
MAX_PRIO, *IDLE_P*
PROPERTIES
$PRIORITY = 0..(MAX\_PRIO - 1) \wedge$
$MAX\_PRIO > 0 \wedge IDLE\_P = 0$
VARIABLES
prio

INVARIANT
$prio \in TASK \pfun PRIORITY \wedge$
$\mathrm{dom}(prio) = tasks \wedge$
$(active = TRUE \Rightarrow$
$prio(idle) = IDLE\_P \wedge$
$\forall t.(t \in ready \Rightarrow prio(t) \leq prio(running)) \wedge$
$\forall t.(t \in ready \Rightarrow IDLE\_P \leq prio(t)))$

Figura 6.16: Especificação do estado do módulo *Task_r*.

...
CONSTANTS
schedule_p
PROPERTIES
$schedule\_p : (\mathbf{FIN}(TASK) \times (TASK \pfun PRIORITY)) \pfun \mathbf{FIN}(TASK) \wedge$
$schedule\_p = \lambda(tasks, prio) \bullet$
$(tasks : \mathbf{FIN}(TASK) \wedge prio : TASK \pfun PRIORITY \wedge tasks \neq \emptyset \wedge tasks \subseteq \mathrm{dom}(prio)$
$\mid \quad tasks \cap prio^{-1}(\max(prio[tasks])))$
...

Figura 6.17: Especificação da função auxiliar *schedule_p*.

$result \longleftarrow t\_create(priority) =$
PRE
$priority \in PRIORITY \wedge$
$running = TASK\_NULL$
THEN
ANY
task
WHERE
$task \in TASK \wedge$
$task \notin tasks$
THEN
$tasks := tasks \cup \{task\} \parallel$
$prio := prio \cup \{task \mapsto priority\} \parallel$
$ready := ready \cup \{task\} \parallel$
$result := task$
END
END

Figura 6.18: Especificação do refinamento da operação *t_create*.

Na operação *t_create* da figura 6.18, assim como na operação *t_create* da figura 6.4, uma nova tarefa é criada através da substituição **ANY** e é representada pela variável *task*. A particularidade desta operação está na asserção $prio := prio \cup \{task \mapsto priority\}$ que atribui uma prioridade para a nova tarefa criada.

Por último, o refinamento da operação *t_startScheduler*, na figura 6.19, cria uma tarefa ociosa na substituição **ANY**, representada pela variável i. Após isso, para essa tarefa, é adicionada uma prioridade pré-definida pela constante *IDLE_P* em $prio := prio \cup \{i \mapsto IDLE\_P\}$.

```
t_startScheduler =
  BEGIN
    active := TRUE ||
    blocked, suspended := ∅, ∅ ||
    ANY
      i
    WHERE
      i ∈ TASK ∧
      i ∉ tasks
    THEN
      tasks := tasks ∪ {i} ||
      prio := prio ∪ {i ↦ IDLE_P} ||
      idle := i ||
    ANY   t   WHERE
      t ∈ TASK ∧
      (ready = ∅ ⇒ t = i) ∧
      (ready ≠ ∅ ⇒ t ∈ ready ∧
        t ∈ schedule_p(ready, prio))
    THEN
      running := t ||
      ready := (ready ∪ {i}) − {t}
    END
  END
END
```

Figura 6.19: Especificação do refinamento da operação *t_startScheduler*.

6.3 Considerações finais

O objetivo principal dessa especificação foi demonstrar que a modelagem do FreeRTOS através do método B é uma abordagem viável. Desse modo, apesar da sua simplicidade, propriedades importantes do sistema foram constatadas, sendo as principais: a mudança de estado de uma tarefa e o gerenciamento de mensagens em uma fila de mensagens.

Para fazer essa modelagem, foram criadas 1974 linhas de códigos dividas em 8 módulos, gerando 538 obrigações de prova, nas quais 489 foram provadas automaticamente pela ferramenta utilizada, Atelier B, e as restantes, 49, necessitaram da intervessão do especificador para serem provadas. Os número de linhas de código, operações e obrigações de prova de cada módulo podem ser vistos na tabela 6.1.

Módulos	Tamanho		Obrigações de prova			Provas
	Operações	Linhas	B.D.	Teor.	Total	Interativas
Config	0	89	0	0	0	0
Types	0	103	1	1	2	1
Scheduler	5	90	0	0	0	0
Task	12	467	1	219	220	28
Queue	7	231	12	33	45	0
FreeRTOSBasic	19	562	37	46	83	2
FreeRTOS	19	562	43	3	49	0
Task_r	12	432	42	100	142	18
Total	55	1974	136	402	538	49

Tabela 6.1: A tabela apresenta, para cada módulo, o número de operações definidas no módulo, o total de número de linhas(incluindo comentários), o número de obrigações de prova (lemas de boa definição, teoremas, e total), e o número of provas interativas requeridas para garantir a veracidade dos teoremas.

Capítulo 7

Modelagem Atual

A modelagem do capítulo anterior serviu como base para o ínicio da especificação do FreeRTOS, mas, para um trabalho mais completo e passível de incrementação, foi necessário realizar alterações na especificação desenvolvida. Essas alterações visaram simplificar a especificação, para facilitar futuros refinamento, especificar as entidades inicialmente abstraídas, semáforo e mutex, e refinar a especificação desenvolvida.

Seguindo esse raciocíno, a especificação atual do FreeRTOS pode ser divida em seções organizadas da seguinte forma: simplificação da modelagem anterior, seção 7.1; adição de novas entidades seção 7.2; e refinamento da especificação 7.3.

7.1 Simplificação da modelagem anterior

Na especificação anterior, as tarefas foram organizadas nos estados em execução, pronta, bloqueada e suspensa. A modelagem desses estados foi demonstrada na seção 6.1.1. Dessa forma, para garantir a propriedade de que a tarefa em execução deve ter prioridade maior que as tarefas prontas, um futuro refinamento dessa especificação deve verificar a prioridade da tarefa em execução e comparar com as prioridades das tarefas de estado pronta. Além disso, com essa especificação, a cada troca da tarefa em execução, é necessário retirar a nova tarefa do conjunto de tarefas pronta.

O novo modelo criado, simplifica o gerenciamento do estado de uma tarefa abstraindo o conjunto de tarefas pronta e criando apenas um conjunto formado pela união da tarefa em execução com as tarefas de estado pronta. Assim, na especificação do estado da máquina *Task*, retirou-se a variável *ready* e criou-se a variável *runable*, para representar as tarefas de estado pronta e a tarefa em execução, como demonstra a figura 7.1.

A vantagem dessa nova especifição pode ser vista na própria figura 7.1, que especifica que a tarefa ociosa deve estar no estado pronto ou em execução da seguinte forma $idle \in runable$, diferente da $(idle = running \lor idle \in ready)$ na figura 6.3, da especificação anterior. Além disso, nos futuros refinamentos, para garantir que a tarefa em execução tenha prioridade maior ou igual que as do estado pronta, basta solicitar a tarefa de maior prioridade ao conjunto *runable*. Também não é

MACHINE *Task*
VARIABLES
...
tasks,
blocked,
runable,
suspended,
running,
...

INVARIANT
...
$blocked \subset tasks \wedge$
$runable \subset tasks \wedge$
$suspended \subset tasks \wedge$
$runable \cap blocked = \emptyset \wedge$
$blocked \cap suspended = \emptyset \wedge$
$suspended \cap runable = \emptyset \wedge$
$tasks = suspended \cup blocked \cup runable \wedge$
$(active = TRUE \Rightarrow$
$runable \neq \emptyset \wedge$
$running \in runable \wedge$
$idle \in runable \wedge$
$TASK\_NULL \notin tasks)$

Figura 7.1: Especificação nova do estado da máquina *Task*.

necessário, na troca da tarefa em execução, retirar a tarefa do estado *runable*, o que era feito anteriormente com o estado *ready*.

7.1.1 Especificação das operações

Com as alterações no estado da máquina, foi necessário também realizar as modificações necessárias nas operações, que manuseam seus estados. Desse modo, as operações *t_create*, *t_startScheduler t_delayTask* e *t_resume* foram alteradas como demonstra as figuras 7.2, 7.3, 7.4 e 7.5 .

$result \longleftarrow t\_create(priority) =$
PRE
$priority \in PRIORITY \wedge$
$active = TRUE$
THEN
ANY
$task, n\_runable$
WHERE
$task \in TASK \wedge$
$task \notin tasks \wedge$
$n\_runable \subset tasks \wedge$
$n\_runable = runable \cup \{task\} \wedge$
$task \neq TASK\_NULL$
THEN
$tasks := tasks \cup \{task\} \parallel$
$runable := n\_runable \parallel$
CHOICE
skip
OR
$running :: n\_runable$
END $\parallel$
$result := task$
END
END;

Figura 7.2: Especificação nova da operação *t_create*.

Na operação *t_create* da figura 7.2, a prioridade da nova tarefa é passada pelo argumento *priority*. Em seguida, uma tarefa que não esta sendo manuseada pelo sistema é escolhida pela variável *task* e adicionada ao conjunto *runable*. Ao final, através da substituição **CHOICE**, a operação pode decidir em manter a tarefa em

execução ou selecionar uma nova tarefa do conjunto *runable* para entrar em execução (*running* :: *n_runable*), o que não era feito pela antiga *t_create* da figura 6.4.

```
t_startScheduler =
PRE
  active = FALSE
THEN
  active := TRUE ||
  ANY
    idle_task,
    n_runable
  WHERE
    idle_task ∈ TASK∧
    idle_task ∉ tasks∧
    n_runable ⊂ tasks∧
    n_runable = runable∪{idle_task}
  THEN
    idle := idle_task∧
    tasks := tasks∪{idle_task} ||
    runable := n_runable
    running :: n_runable
  END
END;
```

Figura 7.3: Especificação nova da operação *t_startScheduler*.

A nova operação *t_startScheduler* da figura 7.3, responsável por iniciar o sistema, criar a tarefa ociosa, através da variável *idle_task* na substituição **ANY**, acrescentando esta ao conjunto de tarefas gerenciadas pelo FreeRTOS, $tasks := tasks \cup \{idle\_task\}$. Em seguida, o conjunto *runable* é atualizado em $runable := n\_runable$ pela variável *n_runable*, definida como o conjunto *runable* mais a variável *idle_task*. Por último, *running* :: *n_runable* define uma nova tarefa a entrar em execução. Observa-se que, nesse momento, não é necessário subtrair a tarefa *running* do conjunto *runable*, como ocorreu na antiga especificação na figura 6.5.

```
t_delayTask(ticks, task) =
PRE
  task ∈ TASK∧
  active = TRUE∧
  running ≠ idle∧
  ticks ∈ TICK
THEN
  ANY
    n_runable
  WHERE
    n_runable ⊂ tasks∧
    n_runable = runable − {running}
  THEN
    runable := n_runable ||
    running :: n_runable
  END ||
  blocked := blocked∪{running}
END;
```

Figura 7.4: Especificação nova da operação *t_delayTask*.

Para a operação *t_delayTask* da figura 7.4, a tarefa em execução é colocada no estado bloqueada ($blocked := blocked \cup \{running\}$) e retirado do conjunto *runable* ($n\_runable = runable - \{running\}$). Em seguida, uma nova tarefa é colocada em execução através da afirmação *running* :: *n_runable*, finalizando o objetivo da operação de bloquear a tarefa em execução.

t_resume(*rtask*) =
PRE
rtask ∈ *TASK* ∧
rtask ∈ *suspended* ∧
active = *TRUE*
THEN
ANY
n_runable
WHERE
n_runable ⊂ *tasks* ∧
n_runable = *runable* ∪ {*rtask*}
THEN
runable := *n_runable* ||
CHOICE
running :: *n_runable*
OR
skip
END
END ||
suspended := *suspended* − {*rtask*}
END;

Figura 7.5: Especificação nova da operação *t_resume*.

Na nova modelagem da *t_resume* na figura 7.5, a tarefa a ser retornada do estado suspenso é representada por *rtask*. Essa tarefa é adicionada ao estado pronta quando inserida no conjunto *runable*, consequentemente, a mesma é retirada do conjunto de tarefas suspensas (*suspended* := *suspended* − {*rtask*}). Após isso, através da substituição **CHOICE**, uma nova tarefa será colocada em execução ou não. Essa decisão será tratada no refinamento dessa operação, pois, para isso, é necessário saber a prioridade da tarefa retornada, se essa é maior que a tarefa em execução ou não.

7.2 Adição dos novos elementos

Na especificação do capítulo 6, foram abstraídos os elementos semáforos e mutex, os quais são tratados nessa nova especificação. Devido a semelhança dessas entidades como a fila de mensagens, esses elementos foram especificados pela máquina *Queue*, aproveitando estruturas já criadas pelas fila de mensagens, como conjuntos de tarefas bloqueadas por leitura e conjunto de tarefas bloqueadas por escrita. Desse modo, o novo estado da máquina *Queue* ficou como demonstra a figura 7.6.

MACHINE
Queue
...
VARIABLES
queues,
queues_msg,
queues_msg_full,
queues_msg_empty,
semaphores,
semaphores_busy,
semaphores_full,
mutexes,
mutexes_busy,
queue_items,
queue_receiving,
queue_sending,
mutex_holder,
...

Figura 7.6: Novo estado da máquina *Queue*.

Nessa nova modelagem da figura 7.6, a variável *queues* representa o conjunto de todas as filas de mensagens, semáforos e mutexes do sistema. Isso foi feito para abrigar as características comuns à esses três elementos em um só conjunto. Em seguida, tem-se os conjuntos responsáveis pelas filas de mensagens, *queues_msg*, *queues_msg_full* e *queues_msg_empty*. O primeiro conjunto representa todas filas de mensagens do sistema, o segundo representa as filas cheias e o terceiro representa as filas vazias.

Ainda na figura 7.6, os conjuntos *semaphores*, *semaphores_busy* e *semaphores_full* servem para armazenar os semáforos manuseados pelo sistema, indicar os semáforos ocupados e os semáforos com o contador cheio, respectivamente. Por último, as variáveis *mutexes* e *mutexes_busy* informam os mutexes dos sistema e os mutex que estão ocupados, nessa ordem. As variáveis *queue_items*, *queue_receiving*, *queue_sending* e *mutex_holder* serão explicadas em seguida, junto com o invariante da máquina *Queue*.

7.2.1 Invariante da máquina *Queue*

Todos os elementos tratados nessa seção (fila de mensagens, semáforo e mutex) possuem a mesma característica, ter um conjunto de tarefas bloqueadas por tentarem utilizar esses elementos. Desse modo, tal característica deve ser relacionada ao conjunto *queues*, responsável por abrigar os três elementos da máquina. A especificação dessa propriedade é demonstrada na figura 7.7. Assim, nessa figura, as asserções $queue\_receiving \in QUEUE \pfun \mathrm{P}(TASK)$ e $queue\_sending \in QUEUE \pfun \mathrm{P}(TASK)$ relacionam um elemento do conjunto abstrato *QUEUE* como um conjunto de tarefas. Em seguida, as afirmações $queues = \mathrm{dom}(queue\_receiving)$ e $queues = \mathrm{dom}(queue\_sending)$ garantem que todos os elementos de *queues* devem possuir essas relações.

INVARIANT

$\ldots$
$queues \in \mathrm{P}(QUEUE) \land$
$queue\_receiving \in QUEUE \pfun \mathrm{P}(TASK) \land$
$queue\_sending \in QUEUE \pfun \mathrm{P}(TASK) \land$
$queues = \mathrm{dom}(queue\_receiving) \land$
$queues = \mathrm{dom}(queue\_sending) \land$
$\ldots$

Figura 7.7: Parte do invariante da máquina *Queue* responsável pela variável *queues*.

Após a definição da característica comum as três entidades na figura 7.7, a particularidade das filas de mensagens "possuir um conjunto para armazenar suas mensagens"foi modelada pela figura 7.8. Nesta, inicialmente é informado, através da afirmação $queues\_msg \subset queues$, que todos elementos de *queues_msg* herdam as características de *queues*. Em seguida, é criada a função *queue_items*

para relacionar um elemento de *QUEUE* a um conjunto de mensagens e, após isso, em $queues\_msg = \mathrm{dom}(queue\_items)$, é sentenciado que todos os elementos de *queues_msg* devem possuir essa relação. Por último, os conjuntos de filas vazias e filas cheias são colocados como subconjuntos de filas de mensagens em $queues\_msg\_full \subset queues\_msg$ e $queues\_msg\_empty \subset queues\_msg$.

INVARIANT
...
$queues\_msg \subset queues \wedge$
$queue\_items \in QUEUE \nrightarrow \mathrm{P}(ITEM) \wedge$
$queues\_msg = \mathrm{dom}(queue\_items) \wedge$
$queues\_msg\_full \subset queues\_msg \wedge$
$queues\_msg\_empty \subset queues\_msg \wedge$
...

Figura 7.8: Parte do invariante da máquina *Queue* responsável pela fila de mensagens.

A parte do invariante da máquina *Queue* responsável pelo semáforo e mutex é mostrada na figura 7.9. Nela, inicialmente o conjunto *semaphores* é dito como subconjunto de *queues* em $semaphores \subset queues$, para herdar os conjuntos de tarefas bloqueadas pela a entidade. Após isso, os conjuntos *semaphores_busy* e *semaphores_full* são especificados como subconjuntos de *semaphores*.

INVARIANT
...
$semaphores \in \mathrm{P}(QUEUE) \wedge$
$semaphores \subset queues \wedge$
$semaphores\_busy \subset semaphores \wedge$
$semaphores\_full \subset semaphores \wedge$
$mutexes \subset queues \wedge$
$mutex\_holder \in QUEUE \nrightarrow TASK \wedge$
$mutexes = \mathrm{dom}(mutex\_holder) \wedge$
$mutexes\_busy \subset mutexes \wedge$
...

Figura 7.9: Parte do invariante da máquina *Queue* responsável pelas entidades semáforo e mutex.

Por último, ainda na figura 7.9, os elementos mutexes adiquirem os conjuntos de tarefas de *queues* através da asserção $mutexes \subset queues$. Em seguida, a função *mutex_holder* é criada para relacionar os elementos do conjunto *mutexes* a uma tarefa, a tarefa que mantém o mutex. Isso é feito através das afirmações $mutex\_holder \in QUEUE \nrightarrow TASK$ e $mutexes = \mathrm{dom}(mutex\_holder)$. Finalmente, o conjunto de mutexes ocupados, *mutexes_busy*, é colocado como subconjunto dos mutexes do sistema em $mutexes\_busy \subset mutexes$.

7.2.2 Especificação das operações

Nessa nova especificação, as operações da máquina *Queue*, além de controlar as características do elemento fila de mensagens, devem também preocupar-se

com os semáforo e mutex. De início, devido às propriedades semelhantes das entidades tratadas pela máquina, tentou-se criar operações comuns que servicem para as três entidades. Entretanto, as operações ficaram muito grandes e o desenvolvimento, como as obrigações de prova delas, tornaram-se complexas. Devido a isso, foi decido que a forma mais viável seria fazer operações individuais para cada entidade. Com isso, ao total foram criadas dezesseis operações, dez a mais que a especificação do capítulo 6.

Operações da fila de mensagens

Para explicar as novas operações relacionadas a fila de mensagens foram escolhidas *q_queueCreate* e *q_sendItem*. Essas são demonstradas através das figuras 7.10 e 7.11.

$$
\begin{array}{l}
xQueueHandle \longleftarrow q\_queueCreate(uxQueueLength, uxItemSize) = \\
\textbf{PRE} \\
\quad uxQueueLength \in QUEUE\_LENGTH \wedge \\
\quad uxItemSize \in \mathrm{N} \\
\textbf{THEN} \\
\quad \textbf{ANY} \\
\quad\quad pxQueue \\
\quad \textbf{WHERE} \\
\quad\quad pxQueue \in QUEUE \wedge \\
\quad\quad pxQueue \notin queues \\
\quad \textbf{THEN} \\
\quad\quad queues := queues \cup \{pxQueue\} \parallel \\
\quad\quad queues\_msg := queues\_msg \cup \{pxQueue\} \parallel \\
\quad\quad queue\_items := queue\_items \cup \{pxQueue \mapsto \emptyset\} \parallel \\
\quad\quad queue\_receiving := queue\_receiving \cup \{pxQueue \mapsto \emptyset\} \parallel \\
\quad\quad queue\_sending := queue\_sending \cup \{pxQueue \mapsto \emptyset\} \parallel \\
\quad\quad queues\_msg\_empty := queues\_msg\_empty \cup \{pxQueue\} \parallel \\
\quad\quad xQueueHandle := pxQueue \parallel \\
\quad\quad \ldots \\
\quad \textbf{END} \\
\textbf{END};
\end{array}
$$

Figura 7.10: Nova especificação da operação *q_queueCreate*.

Nessa especificação da figura 7.10, a nova fila de mensagens criada é representada pela variável *pxQueue*. Em seguida, esta é adicionada ao conjunto *queues* e *queues_msg*. Após isso, à essa fila, são associados um conjuntos de mensagens ($queues\_msg := queues\_msg \cup \{pxQueue\}$), de fila bloqueadas por leitura ($queue\_receiving := queue\_receiving \cup \{pxQueue \mapsto \emptyset\}$) e de filas bloqueadas por escrita ($queue\_sending := queue\_sending \cup \{pxQueue \mapsto \emptyset\}$). Ao final, essa nova fila é colocada no conjunto de fila vazia, *queues_msg_empty* e retornada pela operação.

Uma operação, um pouco mais complexa que a anterior, é a *q_sendItem*, demonstrada pela figura 7.11. Nela, os parâmetros de entrada são: *pxQueue*, fila que será enviada a mensagem; *pxItem*, mensagem que será colocada na fila; e

```
q_sendItem(pxQueue,
        pxItem, copy_position) =
PRE
  pxQueue ∈ queues_msg ∧
  pxItem ∈ ITEM ∧
  pxItem ∉ queue_items(pxQueue) ∧
  copy_position ∈ COPY_POSITION ∧
  pxQueue ∉ queues_msg_full
THEN
  queue_items(pxQueue) :=
        queue_items(pxQueue) ∪ {pxItem} ||
  IF
    queue_receiving(pxQueue) ≠ ∅
  THEN
    ANY
      n_receiving, n_first
    WHERE
      n_receiving ∈ P(TASK) ∧
      n_first ∈ TASK ∧
      n_receiving =
          (queue_receiving(pxQueue) −
            {first_receiving(pxQueue)}) ∧
      ...
      n_first ∈ n_receiving
    THEN
      queue_receiving(pxQueue) := n_receiving ||
      first_receiving(pxQueue) := n_first
    END
  END ||
  IF
    pxQueue ∈ queues_msg_empty
  THEN
    queues_msg_empty :=
        queues_msg_empty − {pxQueue}
  END ||
  CHOICE
    queues_msg_full := queues_msg_full ∪ {pxQueue}
  OR
    skip
  END
END;
```

Figura 7.11: Nova especificação da operação *q_sendItem*.

copy_position, posição que a mensagem será inserida na fila. Inicialmente, a mensagem passada é colocada no conjunto de mensagens da fila atravé da asserção abaixo .

$$queue\_items(pxQueue) := queue\_items(pxQueue) \cup \{pxItem\}$$

.

Após isso, dentro do primeiro **IF** da figura 7.11, é verificado se existem tarefas bloqueadas por leitura da fila. Caso exista, esta é retirada da fila, pois uma nova mensagem será adicionada à fila. Em seguida, no segundo **IF**, é conferido se *pxQueue* está no conjunto de fila vazia, sendo verdadeiro, ela é retirada desse conjunto. Por último, o **CHOICE** possibilita o fato da fila ser inserida no conjunto de filas cheias ou não. Assim, em refinamentos, se para *pxQueue* ficar cheia resta apenas um iten, ela será incluida no conjunto de filas cheias, caso contrário, não será.

Além das operações aqui demonstradas, outras responsáveis por manusear a abstração fila de tarefas são *q_queueDelete*, responsável por excluir uma fila do sistema, e *q_receivedItem*, que retira uma mesagem da fila de mensagens.

Operações do semáforo

No elemento semáforo, duas das principais operações são *q_createSemaphore*, responsável por criar um novo semáforo, e *q_giveSemaphore*, que preocupa-se com a liberação do semáforo ocupado por uma tarefa. Estas são demonstradas nas figuras 7.12 e 7.13.

As especificação da operação *q_createSemaphore*, figura 7.12, é parecida com a da operação *q_queueCreate*, figura 7.10. Nela, são passados como parâmetros a quantidade de tarefas que podem reter o semáforo, ou seja, o contador do semáforo, *maxCount*, e o valor inicial do contador, *initialCount*. Assim, essa operação pode criar tanto semáforos com contador como semáforos binários. Para este último, basta atribuir que o contador do semáforo é 1(um) e inicia-lo com zero.

$$
\begin{array}{l}
rSemaphore \leftarrow q\_createSemaphore(maxCount, initialCount) = \\
\textbf{PRE} \\
\quad maxCount \in QUEUE\_LENGTH \wedge \\
\quad initialCount \in QUEUE\_QUANT \wedge \\
\quad initialCount \leq mathit{maxCount} \\
\textbf{THEN} \\
\quad \textbf{ANY} \\
\quad\quad semaphore \\
\quad \textbf{WHERE} \\
\quad\quad semaphore \in QUEUE \wedge \\
\quad\quad semaphore \notin queues \\
\quad \textbf{THEN} \\
\quad\quad queues := queues \cup \{semaphore\} \parallel \\
\quad\quad queue\_receiving := queue\_receiving \cup \{semaphore \mapsto \emptyset\} \parallel \\
\quad\quad queue\_sending := queue\_sending \cup \{semaphore \mapsto \emptyset\} \parallel \\
\quad\quad semaphores := semaphores \cup \{semaphore\} \parallel \\
\quad\quad \textbf{CHOICE} \\
\quad\quad\quad semaphores\_busy := semaphores\_busy \cup \{semaphore\} \\
\quad\quad \textbf{OR} \\
\quad\quad\quad semaphores\_full := semaphores\_full \cup \{semaphore\} \\
\quad\quad \textbf{OR} \\
\quad\quad\quad skip \\
\quad\quad \textbf{END} \parallel \\
\quad\quad\quad rSemaphore := semaphore \\
\quad \textbf{END} \\
\textbf{END};
\end{array}
$$

Figura 7.12: Operação *q_createSemaphore*.

Ainda na figura 7.12, o novo semáforo criado é representado por *semaphore*. Esse semáforo é colocado inicialmente no conjunto *queues* e, em seguida, são atribuidos a ele um conjunto onde serão armazenadas as tarefas bloqueadas por tentar reter o semáforo ($queue\_receiving := queue\_receiving \cup \{semaphore \mapsto \emptyset\}$) e outro conjunto para as tarefas bloqueadas por tentarem liberar o semáforo ($queue\_sending := queue\_sending \cup \{semaphore \mapsto \emptyset\}$). Por último, através da substituição **CHOICE**, o semáforo pode ser colocado como ocupado ou com o contador cheio.

$q\_giveSemaphore(semaphore) =$
PRE
  $semaphore \in semaphores$
THEN
  CHOICE
    IF
      $semaphore \in semaphores\_busy$
    THEN
      $semaphores\_busy := semaphores\_busy - \{semaphore\}$
    END
  OR
    skip
  END ||
  ANY
    $n\_receiving$,
    n_first
  WHERE
    $n\_receiving \in \mathrm{P}(TASK) \wedge$
    $n\_receiving = queue\_receiving(semaphore) - \{first\_receiving(semaphore)\} \wedge$
    $n\_first \in TASK \wedge$
    $n\_first \in n\_receiving \wedge$
  THEN
    $queue\_receiving(semaphore) := n\_receiving$ ||
    $first\_receiving(semaphore) := n\_first$
  END
END;

Figura 7.13: Operação *q_giveSemaphore*.

Na operação *q_giveSemaphore* da figura 7.13, o semáforo que será liberado é passado como parâmetro. Em seguida, na substituição **CHOICE**, o semáforo pode ser retirado do conjunto de semáforos ocupados ou não, devido a possibilidade do contador do semáforo estar em zero. Por último, na substituição **ANY**, se estiver alguma tarefa bloqueada por leitura do semáforo, ela é retirada do conjunto de tarefas bloqueadas por leitura.

As demais operações relacionadas com o elemento semáforo são *q_deleteSemaphore* e *q_takeSemaphore*. Elas são responsáveis por excluir um semáforo e solicitar a retenção de um semáforo, respectivamente.

Operações do mutex

Na máquina *Queue*, as operações responsáveis pela entidade mutex são *q_createMutex*, *q_takeMutex* e *q_giveMutex*. A *q_createMutex* cria um novo mutex no sistema. Na *q_takeMutex*, um mutex é solicitado. Por último, a *q_giveMutex* libera o mutex retido. As operações *q_createMutex* e *q_takeMutex* são demonstradas na figuras 7.14 e reffig:q$_t$*akeMutex*.

Para criar o novo mutex, a operação da figura 7.14 utiliza-se da substituição *ANY* e da variável *mutex*. O detalhe dessa operação esta na asserção $mutex\_holder :=$

$rMutex \leftarrow q\_createMutex =$
BEGIN
  ANY
    $mutex$
  WHERE
    $mutex \in QUEUE \wedge$
    $mutex \notin queues$
  THEN
    $queues := queues \cup \{mutex\} \parallel$
    $mutexes := mutexes \cup \{mutex\} \parallel$
    $queue\_receiving := queue\_receiving \cup \{mutex \mapsto \emptyset\} \parallel$
    $queue\_sending := queue\_sending \cup \{mutex \mapsto \emptyset\} \parallel$
    $mutex\_holder := mutex\_holder \cup \{mutex \mapsto TASK\_NULL\} \parallel$
    $rMutex := mutex$
  END
END;

Figura 7.14: Operação *q_createMutex*.

$mutex\_holder \cup \{mutex \mapsto TASK\_NULL\}$ que, como o mutex não está retido, é associada uma tarefa nula ao mutex, representando que nenhuma tarefa retém o mutex. Ao final, o mutex criado é retornado pelo parâmentro de saída.

$q\_takeMutex(mutex, task) =$
PRE
  $mutex \in mutexes \wedge$
  $mutex \notin mutexes\_busy \wedge$
  $task \in TASK \wedge$
  $task \neq TASK\_NULL$
THEN
  $mutexes\_busy := mutexes\_busy \cup \{mutex\} \parallel$
  $mutex\_holder(mutex) := task$
END;

Figura 7.15: Operação *q_takeMutex*.

A operação *q_takeMutex* da figura 7.15 é bastante simples. Ela é responsável por atribuir um mutex, *mutex*, à tarefa que obteve sucesso na solicitação deste, *task*. Para isso, ela informa que o mutex está ocupado em $mutexes\_busy := mutexes\_busy \cup \{mutex\}$ e depois associa a tarefa ao mutex, $mutex\_holder(mutex) := task$.

Operações de propósito geral

Além das operações comentadas nessa seção, a máquina *Queue* possui operações de propósito geral, que também são utilizadas pelas funcionalidades do FreeRTOS. Um exemplo dessas operações pode ser visto na figura 7.16, usada pela funcionalidade *vTaskEndScheduler* para reiniciar todos os estados da máquina *Queue*, colocando um conjunto vazio em todas as variáveis da máquina.

```
q_endScheduler =
BEGIN
  queues := ∅ ||
  queues_msg := ∅ ||
  queues_msg_empty := ∅ ||
  queues_msg_full := ∅ ||
  queue_items := ∅ ||
  queue_receiving := ∅ ||
  queue_sending := ∅ ||
  semaphores := ∅ ||
  semaphores_busy := ∅ ||
  mutexes := ∅ ||
  mutexes_busy := ∅ ||
  mutex_holder := ∅ ||
  semaphores_full := ∅ ||
  ...
END;
```

Figura 7.16: Operação *q_endScheduler*.

7.3 Refinamento da especificação

O refinamento dessa nova especificação foi feito de dois modos. Para continuar com a propriedade de prioridade de uma tarefa, o refinamento *Queue_r* foi adaptados para as novas alterações da máquina *Task* e, para tornar esse trabalho menos abstrato, o refinamento da máquina *Queue* foi feito utilizando estruturas de dados mais concretas. Desse modo, o refinamento desse trabalho pode ser divido em refinamento da máquina *Task* e refinamento da máquina *Queue*.

7.3.1 Refinamento da máquina *Task*

Devido as alterações no conjunto da máquina *Task* e a adição de novas abstrações, o refinamento da módulo *Task* ficou mais complexo que o da seção 6.2. Ele começa relacionando todas as tarefas com uma prioridade atual e, devido ao mecanismo de herança de prioridade do mutex, também associa todas as tarefas a uma prioridade base, que será explicada no decorrer dessa seção. Assim, o novo estado do refinamento *Task_r* ficou como demonstra a figura 7.17.

O novo invariante de *Task_r* na figura 7.17 começa declarandos as variáveis *tasks*, *blocked*, *runable*, *suspended*, *running* e *idle*, as quais já foram explicadas na seção 7.1. Após isso, *t_priority* e *t_bpriority*, indicam, respectivamente, as prioridades atual e a prioridade base das tarefas. A associação das tarefas do sistema à uma prioridade atual e feita pelas asserções $t\_priority \in TASK \leftrightarrow PRIORITY$ e $\mathrm{dom}(t\_priority) = tasks$, o mesmo é feito em relação a prioridade base utilizando as afirmações $t\_bpriority \in TASK \leftrightarrow PRIORITY$ e $\mathrm{dom}(t\_bpriority) = tasks$.

Com essas alterações, as operações refinadas támbem foram modificadas. Assim o antigo refinamento da operação *t_create* mostrado na figura 6.18 e modifi-

REFINEMENT *Task_r*
ABSTRACT_VARIABLES
...
tasks,
blocked,
runable,
suspended,
running,
idle,
t_priority,
t_bpriority,

INVARIANT ...
$t\_priority \in TASK \nrightarrow PRIORITY \wedge$
$\mathrm{dom}(t\_priority) = tasks \wedge$
$t\_bpriority \in TASK \nrightarrow PRIORITY \wedge$
$\mathrm{dom}(t\_bpriority) = tasks \wedge$
$(active = TRUE \Rightarrow$
$t\_priority(idle) = IDLE\_PRIORITY \wedge$
$t\_priority(running) = \max(t\_priority[runable]) \wedge$
$IDLE\_PRIORITY \leq \min(t\_priority[tasks])$
)

Figura 7.17: Novo estado do refinamento *Task_r*.

cado para o da figura 7.18. Essa especificação referencia a nova tarefa como *task* e adiciona ela ao conjunto *runable* através da variável *n_runable*. Em seguida, uma prioridade atual é atribuida à tarefa por *n_priority* e a prioridade base é atribuida em $t\_bpriority := t\_bpriority \cup \{task \mapsto priority\}$. De início as prioridades atual e base tem os mesmos valores, o passado como argumento da operação.

Ainda na operação *t_create* da figura 7.18, percebe-se a preocupação com a escolha da tarefa a entrar em execução. Assim, se a prioridade da nova tarefa for maior ou igual que a prioridade da tarefa em execução ($priority \geq t\_priority(running)$) a troca de contexto é realizada, $running :: \mathbf{schedule\_p}(n\_runable, n\_priority)$, escolhendo através da função *schedule_p* uma tarefa dentre as com a maior prioridade do conjunto *n_runable*. Caso a tarefa criada seja a primeira tarefa do sistema ($runable = \emptyset$), ela assume o escalonador imediatamtente ($running := task$).

Para demonstrar o mecanismo de herança de prioridade e a utilização da *t_bpriority* a operação refinada *t_priorityInherit* é demonstrada. Essa operação é utilizada na especificação da funcionalidade *xSemaphoreTake*, responsável por solicitar um mutex. Assim, caso o mutex solicitado esteja ocupado por uma tarefa de menor prioridade, o mecanismo de herança de prioridade é acionado para igualar a prioridade da tarefa que retém o mutex à prioridade da tarefa que solicita o mutex. Ao liberar o mutex, a tarefa retorna a sua prioridade antiga e, para isso, essa prioridade deve ser armazenada na hora da troca.

A especificação da operação da figura 7.19 inicia passando como parâmetro a tarefa que retém o mutex, *holderTask*. A prioridade dessa tarefa é herdada da tarefa em execução, que solicitou o mutex, através da asserções $t\_priority \lhd\!\!+ \{holderTask \mapsto t\_priority(running)\}$ e $t\_priority := n\_priority$. Por fim, a real prioridade da tarefa *holderTask* é armazenada utilizando a asserção $t\_bpriority(holderTask) := t\_priority(holderTask)$.

7.3.2 Refinamento da máquina *Queue*

O refinamento da máquina *Queue*, foi feito através de *Queue_r* e visou três objetivos: tamanho da fila de mensagens; a quantidade de mensagens que a fila pos-

$result \leftarrow t\_create(priority) =$
BEGIN
  ANY $task$, $n\_runable$, $n\_priority$
  WHERE
    $task \in TASK \wedge$
    $task \notin tasks \wedge$
    $n\_runable \subset tasks \wedge$
    $n\_runable = runable \cup \{task\} \wedge$
    $task \neq TASK\_NULL \wedge$
    $n\_priority \in TASK \nrightarrow PRIORITY \wedge$
    $n\_priority = t\_priority \cup \{task \mapsto priority\}$ **THEN**
    $tasks := tasks \cup \{task\} \parallel$
    $runable := n\_runable \parallel$
    $t\_priority := n\_priority \parallel$
    $t\_bpriority := t\_bpriority \cup \{task \mapsto priority\} \parallel$
    IF $active = TRUE$ **THEN**
      IF $priority \geq t\_priority(running)$ **THEN**
        $running ::$ **schedule_p**$(n\_runable, n\_priority)$
      END
    ELSE
      IF $runable = \emptyset$ **THEN**
        $running := task$
      ELSE
        IF $priority \geq t\_priority(running)$ **THEN**
          $running ::$ **schedule_p**$(n\_runable, n\_priority)$
        END
      END
    END $\parallel$
    $result := task$
  END
END;

Figura 7.18: Novo refinamento da operação *t_create*.

sui; e mudança da especificação do conjunto de itens para uma sequência de itens. Esse refinamento foi iniciado acrescentando as variáveis *queue_size*, *queue_quant* e *queue_items_r*. A primeira relaciona uma fila de mensagens a um tamanho, a segunda representa a quantidade de mensagens que a fila possui e a última refina o conjunto *queue_items* como uma sequência de mensagens. Desse modo, o invariante do refinamento ficou como demonstra a figura 7.20.

Com as novas propriedades, as operações da maquina *Queue_r* tornaram-se diferentes das máquinas *Queue*. Um exemplo dessa alteração pode ser visto na figura 7.21. Nela, na criação de uma nova tarefa, são adicionadas as características de quantidade de mensagens e tamanho da fila. Outro detalhe é a relação da nova fila com uma sequência de itens ($queue\_items\_r := queue\_items\_r \cup \{pxQueue \mapsto \emptyset\}$), diferente do conjunto de itens da figura 7.10.

$t\_priorityInherit(holderTask, xTicksToWait) =$
BEGIN
  ANY
    $n\_runable$,
    $n\_priority$
  WHERE
    $n\_runable \subset tasks \wedge$
    $n\_runable = runable - \{running\} \wedge$
    $n\_runable \neq \emptyset \wedge$
    $n\_priority \in TASK \rightarrow PRIORITY \wedge$
    $n\_priority = t\_priority \Leftarrow \{holderTask \mapsto t\_priority(running)\}$
  THEN
    $runable := n\_runable \parallel$
    $blocked := blocked \cup \{running\} \parallel$
    IF $t\_priority(running) > t\_priority(holderTask)$
    THEN
      $t\_priority := n\_priority \parallel$
      $t\_bpriority(holderTask) := t\_priority(holderTask) \parallel$
      $running ::$ **schedule_p**$(n\_runable, n\_priority)$
    ELSE
      $running ::$ **schedule_p**$(n\_runable, t\_priority)$
    END
  END
END;

Figura 7.19: Refinamento da operação *t_priorityInherit*.

REFINEMENT
  Queue_r
INVARIANT
...
  $queue\_size \in QUEUE \rightarrow QUEUE\_LENGTH \wedge$
  $queue\_quant \in QUEUE \rightarrow QUEUE\_QUANT \wedge$
  $queue\_items\_r \in QUEUE \rightarrow iseq(ITEM) \wedge$
  $\mathrm{dom}(queue\_size) = queues \wedge$
  $\mathrm{dom}(queue\_quant) = queues \wedge$
  $\mathrm{dom}(queue\_items\_r) = \mathrm{dom}(queue\_items) \wedge$
...

Figura 7.20: Invariante do refinamento *Queue_r*.

$xQueueHandle \leftarrow q\_queueCreate(uxQueueLength, uxItemSize) =$
BEGIN
  ANY
    $pxQueue$
  WHERE
    $pxQueue \in QUEUE \wedge$
    $pxQueue \notin queues$
  THEN
    $queues := queues \cup \{pxQueue\} \parallel$
    $queues\_msg := queues\_msg \cup \{pxQueue\} \parallel$
    $queue\_items\_r := queue\_items\_r \cup \{pxQueue \mapsto \emptyset\} \parallel$
    $queue\_size := queue\_size \cup \{pxQueue \mapsto uxQueueLength\} \parallel$
    $queue\_receiving := queue\_receiving \cup \{pxQueue \mapsto \emptyset\} \parallel$
    $queue\_sending := queue\_sending \cup \{pxQueue \mapsto \emptyset\} \parallel$
    $queue\_quant := queue\_quant \cup \{pxQueue \mapsto 0\} \parallel$
    $queues\_msg\_empty := queues\_msg\_empty \cup \{pxQueue\} \parallel$
    $xQueueHandle := pxQueue \parallel$
    ...
  END
END;

Figura 7.21: Refinamento da operação *q_queueCreate*.

Capítulo 8

Considerações Finais

Esse trabalho baseia-se em dois grandes desafios da computação, desenvolvimento de sistemas fidedignos e, principalmente, o projeto do software verificado, no qual se encontra a proposta de especificar formalmente o sistema operacional de tempo real FreeRTOS, objetivo principal desse trabalho.

Para esta especificação desse sistema, devido a sua proximidade com as linguagens de programação atual, o formalismo escolhido foi o Método B. Entretanto, no decorre dessa especificação notou-se algumas limitações proporcionada por esse formalismo, o que será discutido na seção 8.2.

Devido a extensão do sistema, as limitações do método B e para torna possível a realização desse trabalho, dentro do prazo de conclusão, algumas características do sistema foram abstraídas. Ao final, as entidades tarefa, fila de mensagens, semáforos e mutexes foram tratadas de forma abstrata.

Como decisão de projeto uma modelagem inicial, mais genérica, foi criada inicialmente para verificar a viabilidade da especificação e constatar novas limitações de escopo. Com essa especificação, percebeu-se que para a extensão dessa especificação era necessário realizar alterações na mesma. Essas alterações foram realizadas através de uma nova modelagem que, além de tornar a anterior mais facilmente expansível, adicionou novas abstrações e junto com elas, novas propriedades. Após isso, como objetivo comprovar o refinamento do modelo criado, um nível de abstração a mais foi desenvolvido na nova especificação.

Ao final, o trabalho aqui desenvolvido, além de ser um esforço inicial para a verificação e especificação do FreeRTOS, representa uma documentação do sistema do ponto de vista forma, diferente da documenta atualmente disponível. Em adição, a especificação aqui criada pode ser usada para verificação de aplicações criadas com o FreeRTOS, como será comentando na seção 8.1. Por último a extensão desse trabalho pode proporcionar a criação de um sistema formalmente verificado baseado no FreeRTOS .

8.1 Trabalhos futuros

Devido à natureza pioneira desse trabalho, uma variedade de esforços podem ser iniciados a partir dele, com o objetivo de complementa a especificação e va-

lidação do FreeRTOS. Assim, alguns trabalhos futuros de suplemento a este são sugeridos a seguir.

Nesse trabalho, para tornar possível a sua especificação, apenas partes especificas do FreeRTOS foram tratadas. Entre as características abstraídas dessa modelagem está a entidade co-rotina e o controle de estouro de tempo proporcionado pelo escalonador. Assim, como trabalhos futuros, pode-se ter a especificação dessas características seguindo a mesma abordagem utilizada nesse trabalho, o que tornaria a modelagem do FreeRTOS mais completa.

Outra possibilidade de extensão desse trabalho está no refinamento da especificação criada. O nível de abstração utilizado nesse trabalho foi relativamente alto. Com isso, como continuidade dessa especificação pode-se complementar, através das técnicas de refinamento do método B, a proximidade dessa especificação com as linguagem de programação, retirando estruturas abstratas, como conjuntos, e colocando estruturas concretas como seqüências e vetores. Parte dessa proposta, já foi iniciada nessa modelagem com o refinamento da máquina *Queue*, demonstrada na seção 7.3.2.

Além dos trabalhos listados acima, outro esforço que pode ser iniciado a partir dessa modelagem é a utilização da especificação criada para a verificação da atual implementação. Isso, é possível através de ferramentas como FRAMA-C [**?**] e VCC [**?**], sendo necessário adaptar a especificação criada para particularidades específicas de cada ferramenta.

Por fim, um trabalho mais simples, mas que agregar valor à especificação aqui desenvolvida, pode-se ajustar a modelagem para a ferramenta de animação e verificação da especificação em B, o ProB[**?**]. Assim, uma animação para a modelagem será desenvolvida e uma nova verificação da especificação pode ser feita utilizando outra ferramenta, podendo analizar outras classes de propriedade através de *model checking*.

8.2 Limitações da método B

Grande parte da abstração do sistema a ser especificado foi devido a limitações do método B. Um problema, enfrentado inicialmente na especificação foi a impossibilidade de verificar problemas conhecidos dos sistemas operacionais como o *deadlock*, que causa bloqueio entre tarefas que aguardam os recursos retidos entre elas. Isso ocorre, devido ao método B não ter mecanismo para verificar situações de concorrência, o que é suprido por outros formalismos como Circus e CSP [**?**].

Além disso, a implementação das atividades realizadas pelas operações também não é possível pelo método B. Para isso, é necessário relacionar uma função a uma tarefa como, por exemplo, a função da figura 8.1 pode ser relacionada a uma nova tarefa criada. Essa técnica é específica das linguagens de programação como C e C++, o que torna difícil a sua especificação. Consequentemente, a estrutura na qual a função de uma tarefa deve obedecer também não pode ser verificada utilizando o método B.

```
void functionName( void *vParameters )
{
      for( ;; )
      {
          -- Task application code here. --
      }
}
```

Figura 8.1: Função executada quando a tarefa associada a ela entrar em execução

Outro problema encontrado durante o desenvolvimento do trabalho, foi a natureza paralela de execução das substituições que compõem uma operação e o não reaproveitamento, na mesma máquina, das operações especificadas.

No método B, as atribuições de uma operação ocorrem em paralelo, assim se em uma operação eu tiver as seguintes seqüências, nessa mesma ordem, $a := a + 1$ e $b := a$, o valor das variável b será o valor da variável a antes do incremento $a + 1$. Isso dificulta a especificação de um sistema, no qual, diferente do paralelismo, suas atividades ocorrem em seqüência.

Uma forma de contornar a limitação acima, que foi muito utilizada nesse trabalho, seria declarar variáveis com os futuros estados da operação após a sua execução, como demonstra a figura 8.2,o que torna a especificação mais extensa. Assim para a variável b do parágrafo anterior assumir o valor incrementado de a é criada uma variável t para representar o valor de $a + 1$

ANY
t
WHERE
$t = a + 1$
THEN
$a = t \parallel$
$b = t$
END

Figura 8.2: Técnica que contorla as atribuições paralelas em B.

O não reaproveitamento das operações especificadas na mesma máquina, força a especificação ficar com partes iguais da modelagem distribuídas por todas as operações, enquanto que, poder-se-ia juntar partes repetidas de código em operações separadas e estas serem reaproveitadas por toda a máquina.

Por último, devido ao método B ser direcionado à aplicações críticas, onde a alocação de memória dinâmica é proibida torna difícil a especificação de estruturas como ponteiros, bastante utilizada no FreeRTOS e não tratada nesse trabalho.

Referências Bibliográficas

[1] COMMITTEE, P. I. T. A. *Computational Science: Ensuring America's Competitiveness*. 2005. URL:http://www.nitrd.gov/pitac/reports/20050609computational/computational.pdf.

[2] SBC. *Grandes Desafios da Pesquisa em Computacão no Brasil 2006*. [S.l.], 2006.

[3] Jim Woodcock et al. Formal methods: Practice and experience. In: ACM (Ed.). *ACM Computing Surveys*. [S.l.: s.n.], 2009. v. 5, p. 1–40.

[4] J.C. Bicarregui; C.A.R. Hoare; J.C.P. Woodcock. The verified software repository:a step towards the verifying compiler. In: *Formal Aspects of Computing*. [S.l.]: Springer, 2008. v. 1, p. 277–280.

[5] Tony Hoare; Jay Misra. *Verified software: theories, tools, experiments*. 2005. URL:http://vstte.ethz.ch/pdfs/vstte-hoare-misra.pdf.

[6] Richard Barry. *Using the FreeRTOS Real Time Kernel - A Pratical Guide*. [S.l.]: FreeRTOS.org, 2009.

[7] Richard Barry. *FreeRTOS Reference Manual - A Pratical Guide*. [S.l.]: FreeRTOS.org, 2009.

[8] BARRY, R. *FreeRTOS Real Time Kernel*. 2010. URL:http://sourceforge.net/projects/freertos/files/FreeRTOS/.

[9] SCHNEIDER, S. *The b-method: an introdution*. [S.l.]: Palgrave, 2001.

[10] ABRIAL, J. R. *The B-book: assigning programs to meanings*. [S.l.]: Cambridge University Press, 1996.

[11] E. Jaffuel; B. Legeard. Leirios test generator: Automated test generation from b models. In: *The 7th International B Conference*. [S.l.: s.n.], 2007. p. 277–280.

[12] Valério Medeiros Jr.; Stephenson Galvão. Modelagem de micro controladores em b. In: *Anais do VIII Encontro Regional de Matemática Aplicada e Computacional (ERMAC)*. [S.l.]: EUFRN, 2008. v. 1.

[13] Bartira Dantas et al. Proposta e avaliacao de uma abordagem de desenvolvimento de software fidedigno por construcao com o método b. In: *Anais do XXVIII Congresso da SBC*. [S.l.: s.n.], 2008.

[14] Bartira Dantas et al. Applying the b method to take on the grand challenge of verified compilation. In: *Brazilian Symposium on Formal Methods (SBMF2008) - Proceedings*. [S.l.]: Editora Gráfica da UFBA - EDUFBA, 2008. v. 1.

[15] David Déharbe; Stephenson Galvão; Anamaria Moreira. Formalizing freertos: First steps. In: *Brazilian Symposium on Formal Methods (SBMF2009) - Proceedings*. [S.l.]: Editora Gráfica da UFRGS - EDUFRGS, 2009. v. 1.

[16] POPEK, G. J. et al. Ucla secure unix. *Managing Requirements Knowledge, International Workshop on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 355, 1979.

[17] FEIERTAG, R. J.; NEUMANN, P. G. The foundations of a provably secure operating system (psos). In: *IN PROCEEDINGS OF THE NATIONAL COMPUTER CONFERENCE*. [S.l.]: AFIPS Press, 1979. p. 329–334.

[18] BEVIER, W. R. *Kit: A Study in Operating System Verification*. 1989.

[19] ROEVER, W.-P. d.; ENGELHARDT, K. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. New York, NY, USA: Cambridge University Press, 2008. ISBN 9780521103503.

[20] KLEIN, G. Operating system verification — an overview. *Sādhanā*, Springer, v. 34, n. 1, p. 27–69, fev. 2009.

[21] HOHMUTH, M.; TEWS, H.; STEPHENS, S. G. Applying source-code verification to a microkernel: the vfiasco project. In: *EW 10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*. New York, NY, USA: ACM, 2002. p. 165–169.

[22] SHAPIRO, J. et al. Towards a verified, general-purpose operating system kernel. In: *In Klein [10*. [S.l.: s.n.], 2004. p. 1–19.

[23] KOLANSKI, R.; KLEIN, G. Formalising the L4 microkernel API. In: JAY, B.; GUDMUNDSSON, J. (Ed.). *Computing: The Australasian Theory Symposium (CATS 06)*. Hobart, Australia: [s.n.], 2006. (Conferences in Research and Practice in Information Technology, v. 51), p. 53–68.

[24] Moritz Kleine; Steffen Helke. Météor: A successful application of b in a large project. In: *World Congress on Formal Methods*. [S.l.]: Springer, 1999. v. 1708.

[25] Iain D. Craig. *Formal Models of Operating System Kernels*. 1. ed. [S.l.]: Springer, 2006.

[26] David Kalinsky. *Basic concepts of real-time operating systems*. 2003.

[27] Qing Li; Carolyn Yao. *Real-Time Concepts for Embedded Systems*. [S.l.]: CMP Books, 2003.

[28] Honeywell. *Formal Method: analysis of complex systems to ensure correctenes and reduce cost*. [S.l.], 2005.

[29] ClearSY. *atelieb manuel-reference*. 2002. URL:http://www.atelierb.eu.

[30] P. Behm et al. Low-level code verification based on csp models. In: *Brazilian Symposium on Formal Methods (SBMF2009) - Proceedings*. [S.l.]: Editora Gráfica da UFRGS - EDUFRGS, 2009. v. 1.

[31] ClearSY. *Atelier B 4.0*. 2009. URL:http://www.atelierb.eu.

[32] CEA LIST. *Frama-C value analysis plug-in*. 2009. URL:http://frama-c.cea.fr/.

[33] Markus Dahlweid Michael Moskal, T. S. *VCC: Contract-based Modular Verification of Concurrent C*. [S.l.], 2008.

[34] LEUSCHELAND, M.; BUTLER1, M. Prob: A model checker for b. In: *FME 2003: Formal Methods*. [S.l.]: Springer, 2003. p. 855–874.

[35] HOARE, C. A. R. *Communicating Sequential Processes*. 2004.