UNIVERSIDADE FEDERAL DE PERNAMBUCO
Centro de Informática
Departamento de Sistemas da Computação

Graduação em Engenharia da Computação

Geração de infraestrutura de comunicação para Sistemas Embarcados a Partir de Diagramas UML-ESL

Rafael Teixeira Mendes de Carvalho

Trabalho de Graduação

Recife
05 de dezembro de 2019

UNIVERSIDADE FEDERAL DE PERNAMBUCO
Centro de Informática
Departamento de Sistemas da Computação

Rafael Teixeira Mendes de Carvalho

Geração de infraestrutura de comunicação para Sistemas Embarcados a Partir de Diagramas UML-ESL

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Departamento de Sistemas da Computação da UNIVERSIDADE FEDERAL DE PERNAMBUCO como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Adriano Augusto de Moraes Sarmento*

Recife
05 de dezembro de 2019

Agradecimentos

Agradeço primeiramente a todos os responsáveis pelo conhecimento que me foi passado durante este curso de graduação. Agradeço também aos meus pais, por todo o apoio e confiança, e à minha família, por sempre estar presente. Aos meus amigos que me acompanharam por toda a vida, meu obrigado, vocês são parte essencial de mim. Aos amigos que fiz durante os anos de universidade, ninguém constrói nada sozinho e sem vocês não teria chegado onde estou. Ao meu orientador, por sempre incentivar a construção deste e outros trabalhos. À todos aqueles que me deram oportunidades, viram em mim potencial e qualidade, me forçaram a buscar mais conhecimento e crescer pessoalmente e profissionalmente. Por fim, agradeço cada pessoa especial que passou pela minha vida, me incentivando, apoiando, acreditando, se divertindo ou somente estando ao meu lado, meu muito obrigado.

The highest forms of understanding we can achieve are laughter and human compassion.

—RICHARD P. FEYNMAN

Resumo

A utilização de modelos e componentes no desenvolvimento de sistemas embarcados pode ser uma forma de otimizar o tempo do projetista e diminuir erros nas fases iniciais do projeto. Devido a complexidade dos sistemas, o refinamento dos modelos é demorado e propício a erros. Dessa forma, este trabalho propõe uma ferramenta de geração de uma infraestrutura de comunicação baseada em uma biblioteca extensível de componentes. Utilizando uma modelagem em nível de serviço, detalhes do projeto são abstraídos durante a fase de modelagem e incorporados posteriormente através da infraestrutura proposta, tal abordagem tem como objetivo simplificar o processo de desenvolvimento ao mesmo tempo em que separa os requisitos funcionais e a comunicação entre módulos. Também foi desenvolvido um gerador de código para C++ a partir do profile UML-ESL. Um estudo de caso foi utilizado para validar a infraestrutura de comunicação gerada e demonstrar sua aplicabilidade.

Palavras-chave: modelagem, nível de serviço, TLM, UML-ESL, C++, geração de código

Abstract

The use of models and components for developing embedded systems can be used to optimize the designer time and diminish errors in early project phases. Because of systems complexity, refining models is a long and error prone task. Thus, this work presents a code generator tool capable of generating a communication infrastructure based on an extensible component library. Using a service level model, project details are abstracted during modelling phase and later incorporated through the proposed infrastructure. This approach seeks to simplify the development process and to decouple the functional requisites and the communication between modules. A code generation tool is presented alongside the infrastructure to generate C++ code from the UML-ESL profile. A case study was used to validate and demonstrate the applicability of this work.

Keywords: system modelling, service level, TLM, UML-ESL, C++, code generation

Sumário

Lista de Figuras

CAPÍTULO 1

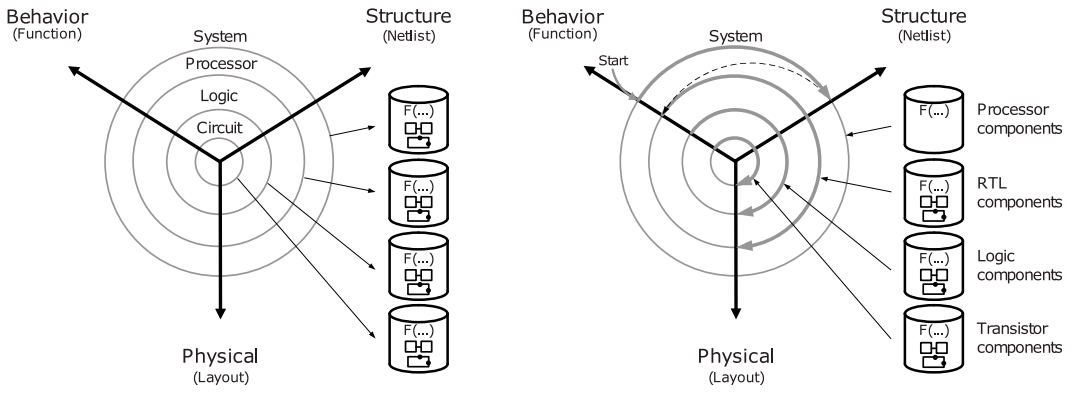
Introdução

1.1 Motivação

O desenvolvimento de sistemas embarcados se torna mais complexo com a adição de novas funcionalidades e protocolos de comunicação a serem implementados. Algumas abordagens são utilizadas para diminuir tal complexidade, como a utilização de novas metodologias, programação baseada em modelos e componentes.

Metodologias de desenvolvimento permitem ao projetista pensar o problema de uma forma diferente, seja a partir de níveis de abstração mais altos, foco em funcionalidades, requisitos ou qualquer parâmetro aplicável. Em [8], Gajski utiliza o Y-Chart (Figura 1.1a) para explicar diferentes metodologias. O Y-Chart é dividido em três eixos, comportamento, estrutura e físico, onde cada eixo possui quatro níveis de abstração, sistema, processador, lógico e circuito. Este modelo é bastante utilizado na indústria de *Very Large-Scale Integration* (VLSI) mas pode ser facilmente entendido para o co-design de hardware e software.

Segundo Gajski, "metodologias de desenvolvimento evoluíram junto à produção tecnológica, complexidade e automação de projetos" (trandução nossa) [8]. Em seu livro, foram apresentadas algumas metodologias, em especial a *bottom-up*, *top-down* e a *meet-in-the-middle*. Uma metodologia *bottom-up* parte dos componentes do sistema em seu nível mais baixo, a junção desses componentes permite o aumento da abstração, tendo por fim o projeto do sistema. Já a *top-down* parte do sentido contrário, iniciando o desenvolvimento por elementos mais abstratos



(a) Y-Chart [8]

(b) *Meet-in-the-middle* [8]

Figura 1.1: Representações de Gajski sobre as metodologias de desenvolvimento

e implementando elementos de baixo nível nas fases finais. A metodologia *meet-in-the-middle* utiliza ambas as técnicas anteriores para "aproveitar as ferramentas disponíveis em baixos níveis de abstração ao mesmo tempo que reduz o layout do projeto em níveis de abstração mais altos" (trandução nossa) [8].

Os modelos aumentam o nível de abstração durante o desenvolvimento além de, segundo Schmidt, "impor restrições específicas e executar verificações de modelo que podem detectar e prever muitos erros ainda no começo do desenvolvimento" (tradução nossa) [20]. Ao projetar em um alto nível de abstração, o problema é simplificado ao diminuir os detalhes de modelagem, dessa forma, simplifica-se também o entendimento e aumenta-se o esforço dedicado às funcionalidades. Kopetz afirma que "projetar voltado à simplicidade significa que devemos construir artefatos onde suas propriedades relevantes possam ser analisadas por modelos simples em diferentes níveis de abstração" (tradução nossa) [10]. Em uma modelagem em nível de serviço ou transação "os detalhes de comunicação entre componentes são separados dos detalhes de funcionamento dos mesmos" (tradução nossa) [2], assim, é possível simplificar os modelos ao focar na comunicação entre os componentes ao invés da funcionalidade do componente em si.

Uma das linguagens mais utilizadas na modelagem de software e sistemas é o *Unified Modeling Language* (UML). A utilização de profiles permite que sejam feitas extensões e customizações adequadas para cada domínio de aplicação. Em sistemas embarcados, profiles como MARTE [23], SysML [22] e UML-ESL [1] oferecem abordagens diferentes de modelagem. Tais profiles de domínio específico permitem ao projetista validar seus projetos ainda em níveis iniciais com ferramentas de automação e verificação.

Usualmente, os modelos utilizam o nível de sistema para representar os elementos que compõem um projeto. Esta representação necessita que sejam especificadas as portas e os canais para a comunicação entre módulos. Uma abordagem diferente é a modelagem em nível de serviço, que abstrai tais detalhes e foca na interação entre os módulos. Quando é utilizado este nível de abstração, a comunicação é separada do funcionamento do sistema e o refinamento para níveis menos abstratos necessita de informações adicionais ou estruturas pré-definidas para preencher as informações necessárias, assim sendo, favorece o reuso de modelos, códigos ou componentes e a divisão de atividades entre equipes.

A comunicação entre sistemas heterogêneos de hardware e software, como FPGAs e microprocessadores é uma tarefa complexa que envolve drivers e protocolos para cada situação. No desenvolvimento baseado em componentes, "o conceito fundamental é que os sistemas sejam construídos pela composição de partes independentes, que são os componentes" [13]. Componentes de comunicação são especialmente importantes pois, ainda que com uma variedade de protocolos e periféricos existentes, camadas de drivers e serviços podem ser reutilizáveis independente do hardware envolvido.

Cada vez mais, a construção de software utiliza ferramentas que aumentam o nível de abstração, seja utilizando metodologias, bibliotecas, frameworks ou modelos. Diversas proposições foram feitas para o domínio de sistemas embarcados, em especial para o desenvolvimento de sistemas heterogêneos de hardware e software com o objetivo de simplificar e agilizar o desenvolvimento de tais sistemas. Ainda assim, existe dificuldade na implantação destes métodos e ferramentas, seja pela alta complexidade na sua utilização ou falta de suporte a modelagem em nível de serviço, assim aumentando o esforço incial dos projetos, seus custos e causando excessivo retrabalho na exploração de possíveis arquiteturas.

1.2 Objetivo e contribuições

A partir do profile proposto por Gomes [1] e Alencar [19], e posteriormente aprimorado por Carvalho [3], este trabalho tem como objetivo o aprimoramento dos diagramas UML-ESL para suportar especificações de plataformas simplificadas, a proposição de uma infraestrutura de comunicação em software para sistemas embarcados, tal como desenvolver um gerador automático de código capaz de fornecer tal infraestrutura baseada em componentes.

Na visão do projetista, o processo de desenvolvimento segue uma metodologia *meet-in-the-middle*, sendo a modelagem em nível de serviço feita em UML-ESL e refinada automaticamente em componentes de nível de processador. A infraestrutura de comunicação em C++ será apresentada em conjunto com uma ferramenta de geração de código, assim como componentes base para uma plataforma. O ciclo de desenvolvimento será demonstrado através de um estudo de caso que engloba todo o processo.

Este trabalho oferece então novas estruturas e ferramentas para a modelagem em nível de serviço que não estão presentes ou estão presentes com precariedade nos modelos existentes. Contribui-se também para a evolução da linguagem UML-ESL, para o desenvolvimento de sistemas embarcados e à engenharia de software como um todo.

1.3 Estrutura do trabalho

Este trabalho segue a seguinte estrutura: no capítulo 2, é discutido a fundamentação teórica sobre modelagem, programação baseada em componentes e geração automática de código. Neste capítulo serão apresentados os principais modelos utilizados atualmente, conceitos relevantes sobre programação de componentes e trabalhos que propuseram geração automática de código a partir de profiles UML. Em seguida, o capítulo 3 é dedicado a explicar as proposições feitas em relação ao profile UML-ESL, assim como a infraestrutura de comunicação e sua geração de código. Serão, então, analisados dois estudos de caso no capítulo 5, da modelagem à implementação. Por fim, o capítulo 6 contém as conclusões deste trabalho e indicações de trabalhos futuros.

CAPÍTULO 2

Fundamentação teórica

Neste capítulo, será discutido o que existe na literatura sobre modelagem, programação baseada em componentes e geração de código automática. Em termos de modelagem, serão apresentadas as linguagens mais utilizadas e suas funcionalidades. Em seguida, serão apresentados alguns conceitos de programação baseada em componentes e um framework capaz de prover a estrutura necessária para este trabalho. Por último, será discutida a geração de código em diversos trabalhos e seus níveis de abstração.

2.1 Modelagem

A modelagem de sistemas é uma área ampla no domínio de sistemas embarcados. Diversos autores propuseram soluções de profiles UML como MARTE [23], SysML [22] e UML-ESL [1][Alencar e Sarmento 2016][3]. Cada profile possui características específicas, SysML, um dos profiles UML mais utilizados na área, "suporta especificação, análise, projeto, verificação e validação de ampla gama de sistemas e sistemas de sistemas" (tradução nossa) [15]. Já MARTE possui um foco maior em "desempenho e propriedades de escalonamento em sistemas de tempo real" (tradução nossa) [15].

As aplicações específicas de cada profile permitem aos projetistas abordar o problema a partir de diferentes aspectos. Segundo Espinoza, "um grande impedimento para qualquer tipo de aplicação no mundo real é o fato de que um único profile pode não ser suficiente para capturar todos os aspectos da multidisciplinaridade no domínio de sistemas embarcados" (tradução nossa) [7].

A modelagem em nível de serviço ou transação (SLM ou TLM) vem sendo utilizada para especificar os aspectos comportamentais da comunicação entre módulos sem se preocupar com as questões funcionais do que cada módulo faz. A Figura 2.1 demonstra os detalhes específicos em cada incremento de detalhes, da modelagem à implementação. Para definir o processo de modelagem, Cai [2] utilizou dois eixos, computação e comunicação, e três níveis de precisão, *un-timed*, *approximated-time* e *cycle-timed*. Utilizando somente um dos eixos por refinamento, é possível adicionar detalhes relevantes que trazem mais precisão ao projeto do sistema. A separação entre os aspectos de computação e de comunicação torna a modelagem em nível de serviço um aliado das equipes pequenas, e de equipes grandes que necessitam dividir o trabalho em etapas menores e mais específicas.

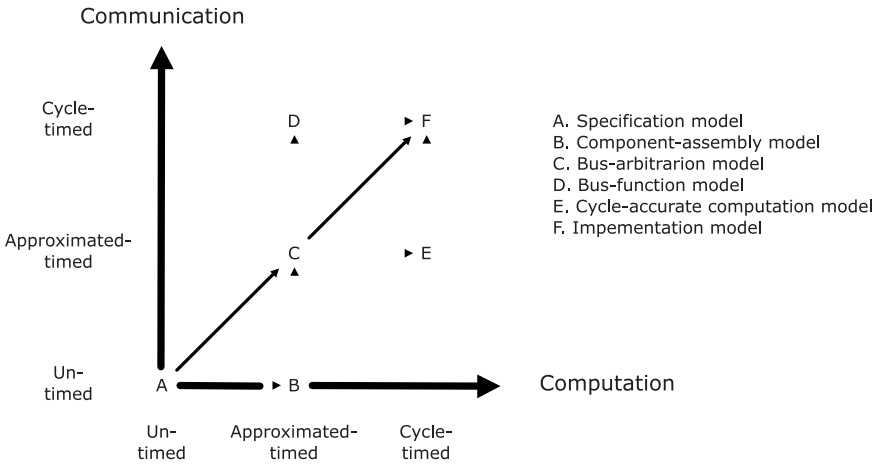
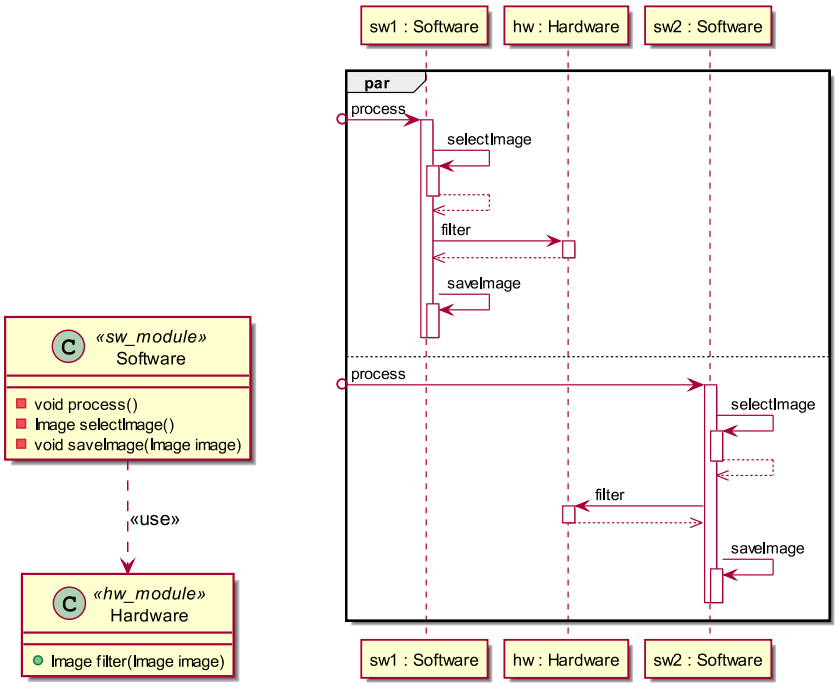


Figura 2.1: Modelagem de sistema [2]

Segundo [9], tanto SysML como MARTE não possuem especificações para a modelagem em nível de serviço. A separação dos aspectos de comunicação dos de computação foi um objetivo buscado por diversos autores, como Parischa [16], Jain [9], Gomes [1], Alencar [19] e Carvalho [3], visando aumentar o nível de abstração nas fases iniciais de projeto.

O UML-ESL foi especificado como um profile UML para modelagem em nível de serviço com foco exclusivo na comunicação. Os diagramas propostos inicialmente em [1] e aperfeiçoados em [19] e [3] oferecem a possibilidade de definir módulos de software e de hardware através do diagrama de classes (Figura 2.2a). Cada módulo pode possuir serviços a serem oferecidos a outros módulos ou métodos internos para auxiliar no fluxo de execução. Também foi definido o diagrama de objetos e de sequência. O diagrama de objetos serve para especificar instância de módulos e propriedades individuais para cada instância ou propriedades de comunicação específicas. Já o diagrama de sequência (Figura 2.2b) define as interações entre os módulos. Tais interações podem ser fluxos de execução que executam chamadas a serviços de forma síncrona ou assíncrona e também aninham múltiplas chamadas em blocos paralelos ou sequenciais. A Figura 2.2 exibe os diagramas de classe e sequência modelados em UML-ESL contendo dois módulos, um de software e outro de hardware. Os módulos de software executam paralelamente o método de processamento, que por sua vez utilizam o serviço disponibilizado pelo módulo de hardware.



(a) Diagrama de classes (b) Diagrama de sequência

Figura 2.2: Exemplo de modelagem em UML-ESL

2.2 Componentes

A programação baseada em componentes é um método de organização do software no qual partes deste software são divididas e desacopladas dos outros módulos, ou seja, “podem ser implantadas de forma independente e compostas sem ser modificadas” [4]. A utilização de tal método ocorre em diversos domínios da engenharia de software, inclusive em sistemas embarcados. Ferramentas comerciais como *STM32CubeMX* [21] e *Matlab/Simulink* [12] oferecem suporte a modelagem, geração de código e/ou componentes a partir de uma biblioteca pré-existente. Tais ferramentas, por serem proprietárias, possuem limitações quanto ao espectro de componentes a serem usados e suas área de aplicação, dessa forma, opções mais flexíveis são necessárias para projetos que necessitam de uma integração maior entre plataformas customizadas ou com elementos diversos.

O framework ESCoRT "trata-se de um ferramental completo para o desenvolvimento de sistemas embarcados utilizando componentes"[13]. É oferecida uma estrutura hierárquica de projeto para sistemas embarcados compostos pelas camadas de abstração de hardware (HAL), kernel, driver, serviço e aplicação. Segundo Melo [13], tal técnica permite uma alta reusabilidade do código, fácil portabilidade e melhor divisão das atividades específicas envolvidas no projeto.

Apesar da construção de uma biblioteca de componentes ser uma tarefa demorada e exaustiva, ela pode ser implementada de forma gradual à medida que novas funcionalidades são necessárias. Enquanto projetos iniciais possuem um tempo maior de desenvolvimento, projetos subsequentes podem ser desenvolvidos mais rapidamente, com menos expertise de hardware e maior foco na aplicação. A partir de abstrações, é esperado mais dedicação aos detalhes e requisitos do sistema, oferecendo assim um produto de melhor qualidade e com menos erros.

No domínio de sistemas embarcados, a possibilidade de se ter uma equipe mista durante o desenvolvimento, com especialistas em microcontroladores/microprocessadores e especialistas em software também é um atrativo da abordagem de componentes. Em especial, o framework ESCoRT separa as camadas de hardware, kernel e driver das camadas de serviço e aplicação, responsáveis pela lógica geral do sistema. Os componentes podem então ser implementados para diferentes microprocessadores ou microcontroladores, com suporte a diferentes sistemas operacionais (Windows, Linux, FreeRTOS, etc.) onde drivers e periféricos de comunicação podem ser encapsulados em interfaces, permitindo total desacoplamento vertical e alta portabilidade das camadas de aplicação e serviço.

2.3 Geração de código

A automatização na geração de código é bastante utilizada para diminuição do esforço do projetista, em especial para tarefas repetitivas. A implementação manual de tais tarefas é altamente suscetível a erros, desmotivador e contra-produtivo. Neste sentido, diversas ferramentas foram desenvolvidas ao longo dos anos para tratar de tal problema. Abordagens diferentes, com propósitos diferentes, oferecem níveis de abstração variados na fase de modelagem e no resultado gerado.

Riccobene [18] propôs um ambiente de desenvolvimento composto por modelagem em diagramas baseados em UML e SystemC e um gerador automático de código para C, C++ ou SystemC. A linguagem de modelagem utilizada possui baixa abstração e, como foi baseada em SystemC, é necessário definir portas, canais e outros elementos que envolvem a comunicação. Os modelos são em nível de sistema, misturando os requisitos funcionais com os processos de comunicação e aumentando a complexidade do projeto.

Yu [24] utilizou *System Level Design Language* (SLDL) como linguagem de modelagem para geração de código C. A partir da especificação são geradas abstrações de tarefas que se comunicam através de barramentos, além de elementos de sincronização e escalonamento. Código C é então gerada para cada elemento de processamento de acordo com as abstrações definidas anteriormente. Por último, elementos adicionais como um RTOS real são adicionados ao projeto para geração do executável. Apesar de suportar a especificação de comunicações, "canais e mecanismos de controle para comunicação são descritos explicitamente" (tradução nossa) [24]. Assim como o modelo de Riccobene, SLDL modela em nível de sistema com baixa abstração para canais, barramentos e outros meios de comunicação.

Utilizando MARTE como linguagem de modelagem, os trabalhos desenvolvidos em [11] e [14] propuseram abordagens diferentes para geração de código. Lennis [11] utilizou uma metodologia baseada em Y-Chart em que há refinamento de um modelo mais abstrato para um menos abstrato através de uma mapeamento de plataforma. A desvantagem dessa abordagem é a necessidade de se definir tal mapeamento de plataforma antes mesmo da geração do código. A falta de suporte de MARTE à modelagem em nível de serviço diminui o nível de abstração ao requerer maiores detalhes sobre os possíveis fluxos de execuções e condições para a execução de eventos. Nicolas [14] propõe uma geração de código para o sistema operacional Linux ou utilizando *Multicore Communications API* (MCAPI) para maior compatibilidade. A solução proposta requer que "o usuário forneça códigos da aplicação em C/C++ que sejam independentes de plataforma e que implementem as funcionalidades de todos os componentes de aplicação" (tradução nossa) [14]. A utilização de componentes na geração e de uma API genérica (MCAPI) permite uma maior flexibilidade na utilização de plataformas distintas, porém, a necessidade de se prover códigos de da aplicação antes da geração do projeto implica em uma alta interdependência entre as funcionalidades e a comunicação.

Com o intuito de gerar componentes de comunicação, Dória [6] propôs um ambiente de geração de componentes para "comunicação entre processos que estão alocados em diferentes processadores" [6]. É necessário a especificação estrutural da plataforma, porém, os componentes utilizados para comunicação são obtidos através de uma biblioteca pré-existente que podem ou não sofrer alterações na fase de geração. A arquitetura proposta é versátil e suporta diversas topologias de rede, porém, a definição de protocolos específicos limita a comunicação com componentes de terceiros que não seguem a mesma arquitetura. Assim como outros trabalhos, a modelagem não é feita em nível de serviço, o que implica em maiores detalhes do sistema a serem especificados.

CAPÍTULO 3

Desenvolvimento

Neste capítulo serão apresentadas as mudanças propostas aos diagramas UML-ESL a fim de suportar as funcionalidades implementadas na infraestrutura de comunicação. Será também apresentada a infraestrutura, as partes que a compõe e seu funcionamento. Por último, será mostrado o processo de geração de código e melhor detalhada a semântica entre UML-ESL e a infraestrutura proposta.

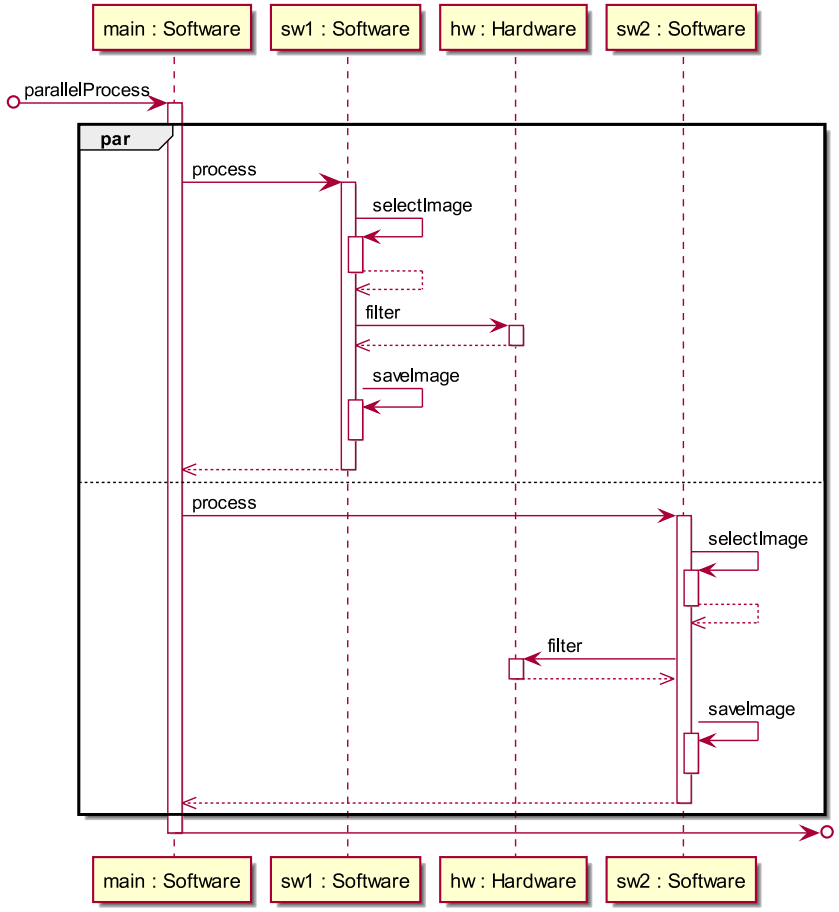
3.1 Diagramas UML-ESL

Os diagramas UML-ESL oferecem uma modelagem em nível de serviço a partir de um profile UML. A versão proposta por Alencar [19] consiste de dois diagramas: classe e sequência. Posteriormente, o conceito de fluxo de execução (*execution flow*) foi proposto por Carvalho [3]. Este novo conceito trouxe a possibilidade de adicionar pontos de entrada (*entry point*) e saída (*exit point*) para criar os fluxos de execução.

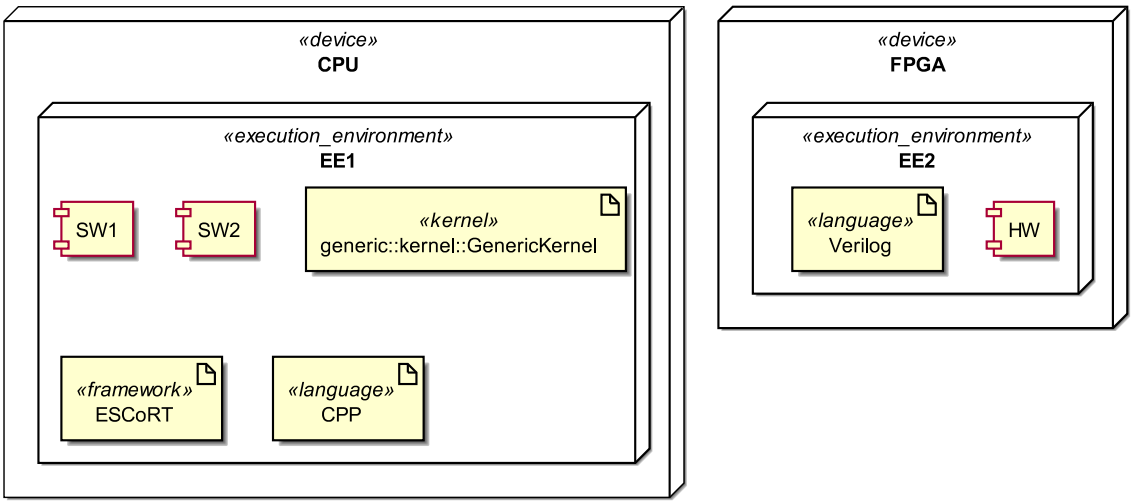
No modelo atual, os fluxos de execução são constantes, organizados de forma paralela ou sequencial e são iniciados através de pontos de entrada sem a possibilidade de serem chamados por elementos externos. Esta abordagem causa um problema ao modelar sub-sistemas e/ou separá-los em hardwares distintos. É proposto então uma alteração no conceito e funcionamento de tais elementos. Os fluxos de execução passam a ser independentes, não possuem execução própria e precisam ser iniciados pelo projetista.

Os fluxos de execução continuam possuindo um ponto de entrada e um ponto de saída obrigatórios, porém, não podem mais estar presentes em blocos sequenciais ou paralelos, já que não possuem execução própria. Por serem independentes, não há conflito nem a necessidade de sincronização entre fluxos de módulos não conectados. Cada ponto de entrada inicia um novo fluxo de execução que deverá estar disponível para ser iniciado a qualquer momento. Uma modelagem semelhante pode ser obtida em comparação com o modelo anterior utilizando um fluxo que contenha um bloco sequencial/paralelo com duas chamadas a serviços. Na Figura 3.1a, o diagrama de sequência feito na Figura 2.2b foi refeito no modelo atual, onde um fluxo de execução é iniciado em um terceiro módulo e, a partir dele, são feitas as chamadas a serviço em paralelo.

Uma segunda proposição deste trabalho em relação aos diagramas UML-ESL é o diagrama de implantação simplificado. A geração de projetos em C++ para plataformas específicas exige



(a) Novo diagrama de sequência UML-ESL



(b) Diagrama de implantação UML-ESL

Figura 3.1: Modificações do profile UML-ESL

a definição de elementos que não eram relevantes no desenvolvimento de simulações, como por exemplo, em qual hardware cada módulo será alocado, quais as características desse hardware e quais interfaces de comunicação estão disponíveis para a interação entre módulos.

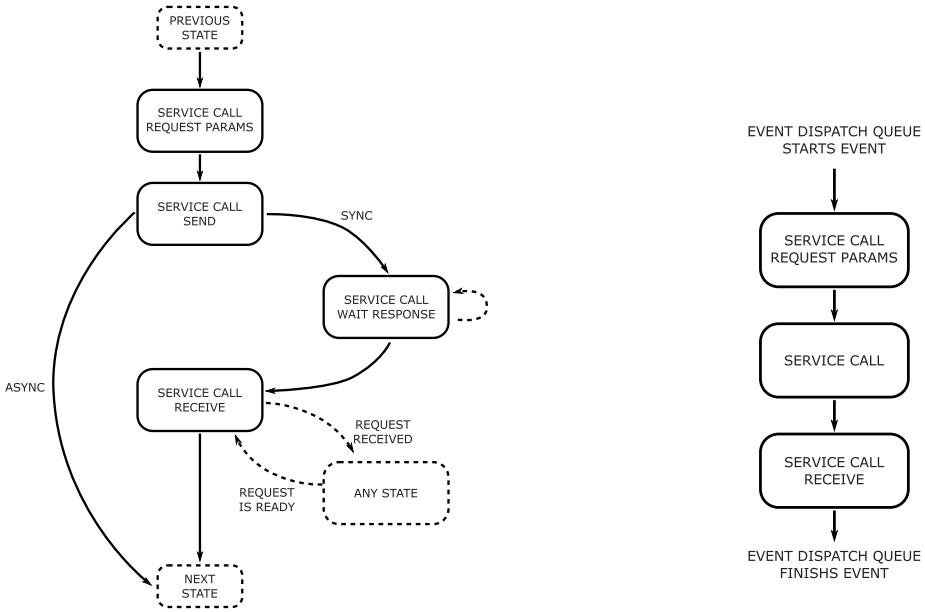
O diagrama de implantação oferece os elementos de *device*, *execution environment*, *component*, *artifact* e *association* para definir as características físicas do sistema. Os *devices* são responsáveis por definir um componente físico, como um microprocessador, microcontrolador, FPGA ou outro. Já os *execution environments* estão contidos nos *devices* e especificam um ambiente de execução que irá possuir características específicas definidas através de *artifacts*. Alguns tipos de *artifacts* válidos são *«kernel»*, *«language»* e *«framework»*, que definem, respectivamente, um sistema operacional a ser utilizado, a linguagem de programação que deseja gerar o projeto e um ou vários frameworks a serem utilizados. É também necessário definir interfaces de comunicação utilizando o elemento de *association* entre dois *devices*. A Figura 3.1b ilustra dois *devices*, a CPU contém um *execution environment* com dois módulos de software (SW1 e SW1), utiliza um kernel genérico, framework ESCoRT e deve ser implementado em C++. O segundo *device* é um FPGA que contém um *execution environment* com um módulo (HW) e deve ser implementado em Verilog.

3.2 Infraestrutura de comunicação

Uma infraestrutura de comunicação é capaz de oferecer mais agilidade no desenvolvimento de produtos, além de qualidade e padronização do código e da arquitetura do sistema, especialmente quando desenvolvido por diversos programadores em equipes diferentes. Este trabalho propõe uma infraestrutura a ser gerada a partir de uma especificação UML-ESL. A infraestrutura utiliza o framework ESCoRT e oferece dois elementos em forma de componentes: componentes personalizados gerados automaticamente e uma biblioteca base.

Os componentes gerados são primeiramente divididos em projetos derivados do diagrama de implantação. Para cada *execution environment* definido, um projeto é gerado, podendo ser de hardware ou software. Cada projeto conterá os módulos associados para serem implementados e também os módulos relacionados (aqueles em que os serviços são utilizados pelos módulos associados).

Os módulos relacionados contêm as chamadas de serviços definidas no diagrama de sequência. A proposta de máquina de estados feita em [3] (Figura 3.2a) foi utilizada parcialmente para a implementação desta infraestrutura. Originalmente, uma máquina de estados completa era composta por diversas submáquinas de acordo com a quantidade de chamadas a serviços. As máquinas iniciavam com a aquisição dos parâmetros e depois era feito o envio e espera dos dados, finalizando com o recebimento da resposta. Enquanto a estrutura funciona bem para chamadas assíncronas, podendo navegar entre os estados de acordo com a necessidade, há uma dificuldade de organizar as chamadas paralelas.



(a) Proposta por Carvalho [3] (b) Estados implementados

Figura 3.2: Estados das chamadas a serviço

Dessa forma, as chamadas a serviço agora possuem máquinas de estados independentes controladas pelo *EventDispatchQueue* (responsável por organizar as execuções paralelas e sequencias, e chamadas síncronas e assíncronas) e contém somente três estados: *request params*, *service call* e *receive* (Figura 3.2b). O estado de *request params* é responsável por obter os parâmetros (dados de entrada do serviço definidos no diagrama de classes) para a chamada ao serviço , enquanto o estado de *receive* por tratar a resposta recebida (dados de saída do serviço definidos no diagrama de classes). Chamadas a serviços que não possuem parâmetros não terão o estado *request params*, e, analogamente, quando não houver resposta, não terão o estado de *receive*. O estado de *service call* executa o serviço em si, seja através de uma chamada remota ou local.

Cada módulo pode implementar um objeto de contexto a ser compartilhado durante uma ou mais chamadas através do método *getExecutionContext*. Quando utilizado, o objeto de contexto pode ser definido pelo usuário através da implementação da interface *IContext* e é utilizado para guardar informações pertinentes ao fluxo de execução atual. Sempre que um fluxo de execução acessa um módulo através de um ponto de entrada ou chamada de serviço este método é chamado para obter uma instância de contexto. Os contextos são limitados ao módulo ao qual estão associados, ou seja, não é possível levar um contexto junto com uma chamada a serviço à outro módulo. O desenvolvedor pode optar por compartilhar um mesmo contexto durante chamadas consecutivas aos serviços do módulo ou criar novos contextos, em ambas as situações, faz-se necessário observar a necessidade de criação de mecanismos adequados de exclusão mútua no acesso ao contexto devido a possíveis paralelismos. O uso dos objetos de contexto é feito nos estados de *request params* e *receive* de forma que é possível

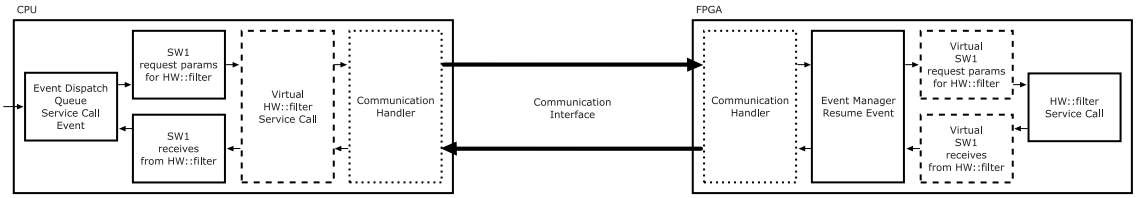


Figura 3.3: Chamadas a serviço entre *devices* diferentes

ao desenvolvedor armazenar os dados recebidos para serem usados em uma chamada a serviço subsequente.

O segundo elemento oferecido nesta infraestrutura é a biblioteca base de componentes, que inclui definições essenciais e é também responsável pela execução e tratamento dos eventos, sejam gerados internamente pelo software ou a partir da comunicação com outros módulos. O tratamento dos eventos é feito pelo componente de kernel chamado *event dispatch queue*.

Uma das definições mais importantes do sistema é o conceito de eventos. Um evento pode ser uma chamada um serviço (*service call*), repetição (*loop*) ou um agrupamento de subeventos e possuem um código único de evento. Os agrupamentos são responsáveis por organizar chamadas paralelas ou sequenciais. Por exemplo, um evento de agrupamento sequencial que possui três subeventos será considerado finalizado quando todos os três subeventos finalizarem sua execução. Eventos de repetição podem executar o mesmo bloco de eventos diversas vezes baseados em uma função de parada a ser definida pelo desenvolvedor.

Eventos de chamadas a serviços são derivados diretamente do diagrama de sequência, podem ser síncronos ou assíncronos, e sempre possuem um módulo que origina a chamada (*caller*) e um módulo que recebe a chamada (*callee*). As chamadas a serviços podem ser simples chamadas a funções ou necessitar de comunicação entre processos ou através de barramentos, dessa forma, eventos entre *devices* diferentes possuem o fluxo de execução quebrado e um novo fluxo de execução é iniciado no dispositivo que está sendo chamado. Esta implementação permite encadear os eventos mesmo em equipamentos distintos ao mesmo tempo que mantém um fluxo de execução semitransparente para o projetista, sendo necessário somente o envio do código de evento durante a comunicação.

A Figura 3.3 ilustra os estados necessários para uma quebra no fluxo de execução da chamada ao serviço *filter* modelado na Figura 3.1. O dispositivo CPU contém o módulo associado SW1 e o módulo relacionado HW, enquanto o dispositivo FPGA contém o módulo associado HW e o módulo relacionado SW1. Ao iniciar o evento de chamada ao serviço, o *EventDispatchQueue* requisita os parâmetros ao módulo SW1 através do estado *request params*, que são então repassados para o serviço. Como o módulo HW é um módulo virtual, é feita a comunicação com o dispositivo externo, que é responsável por capturar e reconstruir o fluxo de execução utilizando o código de evento. No dispositivo FPGA, um módulo virtual SW1 é utilizado na reconstrução do fluxo de execução, o estado de *request params* é chamado para organizar os

dados recebidos da comunicação e então a chamada ao serviço é feita. Com o fim da execução do serviço, é traçado o caminho inverso. Ainda no dispositivo FPGA, o estado virtual *receive* do SW1 é utilizado para organizar os dados e, após o evento finalizado, a resposta da comunicação é retornada para o dispositivo CPU. Neste momento, o serviço virtual *filter* retorna os dados, o estado *receive* do módulo SW1 é chamado e o evento finalizado.

A execução e tratamento dos eventos é feita pelo *event dispatch queue*, que funciona como uma fila que recebe os eventos sendo chamados e atribui o evento a uma thread que irá processá-lo. O processamento do evento inclui as chamadas a serviço, administração de loops e a garantia de execução sequencial ou paralela dos subeventos.

3.3 Geração de código

Com o objetivo de automatizar o refinamento do modelo UML-ESL para a infraestrutura proposta, foi desenvolvido um gerador de código que faz a leitura dos diagramas, interpreta os dados e gera o código em C++. O gerador de código foi desenvolvido em python e utiliza os diagramas UML-ESL modelados através da linguagem PlantUML [17].

O primeiro passo na geração, é a criação de projetos para cada dispositivo especificado no diagrama de implantação. Assim, para cada *execution environment* é criado um projeto com seu nome. Em seguida, para cada projeto, são analisados os componentes associados ao *execution environment* e criados os módulos base e os módulos de usuário (que herdam dos módulos base). A partir dos componentes associados, são extraídos os componentes relacionados, ou seja, aqueles que não estão originalmente presentes naquele *execution environment* mas possuem serviços requisitados pelos módulos associados. Para estes componentes, também são criados, de forma simplificada, módulos base e módulos de usuários. Por fim, é criado o componente *EventManager*, onde estão especificados os eventos com seu código e subeventos.

Os módulos base são classes abstratas e possuem os métodos virtuais de *request params* e *receive* para cada chamada a serviço especificada no diagrama de sequência além de métodos virtuais para cada serviço oferecido que necessite de comunicação com *execution environments* diferentes do atual. Estes métodos devem ser implementados nos módulos de usuário. Além de possuir estes métodos virtuais, os módulos base implementam os serviços internos e o tratamento dos eventos para chamada aos métodos de *request params* e *receive*.

Os eventos são extraídos do diagrama de sequência para criação do *EventManager*. Serão considerados eventos para o projeto atual somente aqueles que se originam ou possuem como destino os módulos associados, como também todos os subeventos até que o fluxo de execução seja quebrado para comunicação com serviços externos. Para cada ponto de entrada, são criados métodos públicos para o início destes fluxos pelo usuário. Um método adicional *void resumeEvent(unsigned int event)* é disponibilizado para a reconstrução dos fluxos de execução após uma comunicação.

CAPÍTULO 4

Estudo de caso

Foram desenvolvidos dois estudos de caso com o intuito de demonstrar a modelagem a partir dos diagramas UML-ESL e a geração e implementação do projeto de software. Foram utilizadas as principais funcionalidades disponíveis como as chamadas síncronas e assíncronas, execuções paralelas e sequenciais, especificação dos módulos e alocação dos recursos.

4.1 Diagnóstico de glaucoma

O primeiro projeto, desenvolvido por Dantas [5], é um sistema que auxilia o diagnóstico de glaucoma implementado em hardware e software e foi portado parcialmente. O sistema utiliza uma imagem do fundo de olho como ilustrado na Figura 4.1 e analisa o dano no nervo óptico através da relação entre o diâmetro do dano ou escavação (*cup*) e o diâmetro do disco óptico (*disc*). A Figura 4.2 ilustra o diagrama de classes em UML-ESL onde foram definidos quatro módulos, dois de software e dois de hardware.

Os módulos de hardware são a câmera e o processamento da imagem em FPGA. O módulo da câmera oferece o serviço de *getFrame* que obtém e retorna um frame da câmera como uma imagem RGB. Já o módulo de processamento em hardware possui dois serviços: *extractCupLimits* e *calculateCupDistance*. A extração dos limites da escavação (*extractCupLimits*) recebe a imagem RGB e retorna os limites inferior e superior como posições em pixel a partir do topo da imagem. Já o cálculo da distância da escavação (*calculateCupDistance*) recebe os limites e calcula o diâmetro, também em pixels, da escavação.

Já os módulos de software são responsáveis por iniciar o processamento de um frame, módulo *Main*, e por processar a imagem a fim de extrair as informações do disco óptico, módulo *SWProcessing*. O módulo principal (*Main*) não oferece serviços, mas possui um método in-

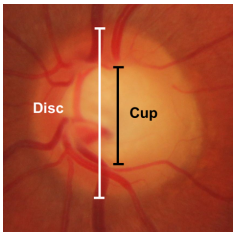


Figura 4.1: Fundo de olho [5]

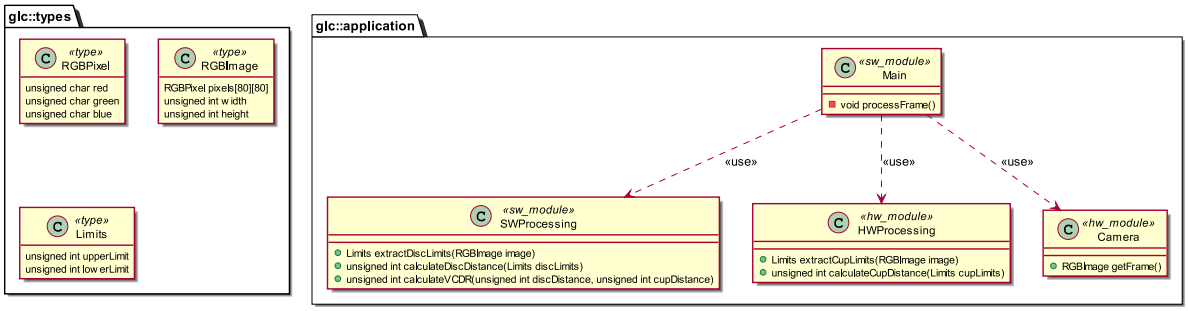
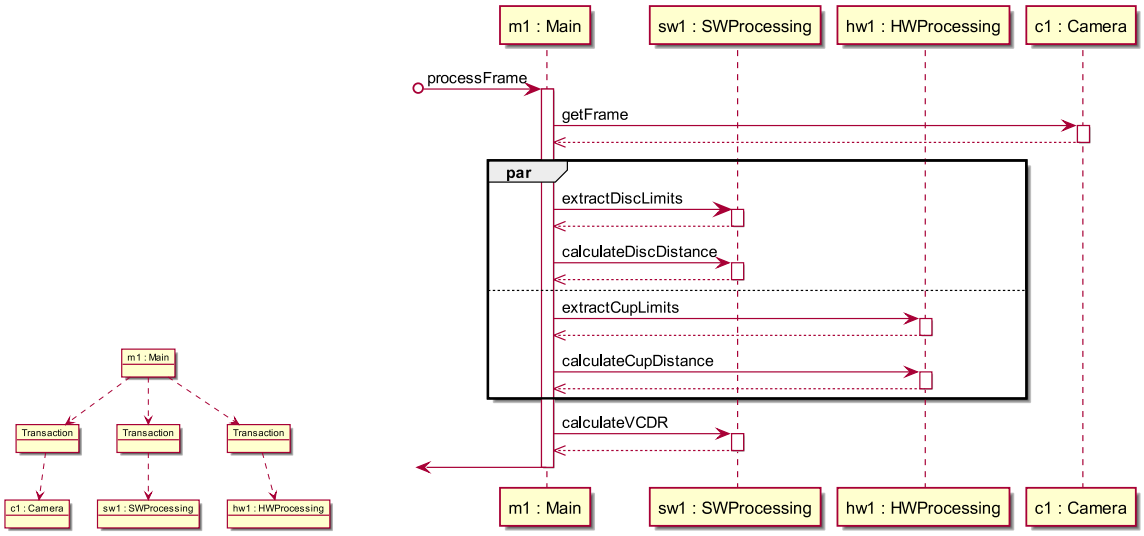


Figura 4.2: Diagrama de classes do projeto Glaucoma



(a) Diagrama de objetos (b) Diagrama de sequência

Figura 4.3: Modelagem do projeto Glaucoma

terno utilizado como ponto de entrada para os eventos. O processamento de software fica a cargo do módulo *SWProcessing* que oferece três serviços. Assim como o módulo de processamento de hardware é responsável por extrair dados da escavação, o módulo de processamento de software extrai dados do disco ocular. Para tal, os serviços *extractDiscLimits* e *calculateDiscDistance* são oferecidos. O primeiro obtém os limites inferior e superior como posições em pixels a partir do topo da imagem, e o segundo obtém o diâmetro do disco. Um terceiro serviço chamado *calculateVCDR* é oferecido para calcular a proporção entre a escavação e o disco (*VCDR - vertical cup disc ratio*).

O diagrama de objetos (Figura 4.3a) declara uma instância de cada módulo, *Main*, *Camera*, *SWProcessing* e *HWProcessing*, além de uma transação entre módulos onde há comunicação. As transações representam o conjunto necessário para uma chamada a serviço, do ponto de vista de quem chama, o que engloba possíveis comunicações entre dispositivos diferentes. Este diagrama tem como finalidade a definição de restrições de projeto para cada módulo ou para

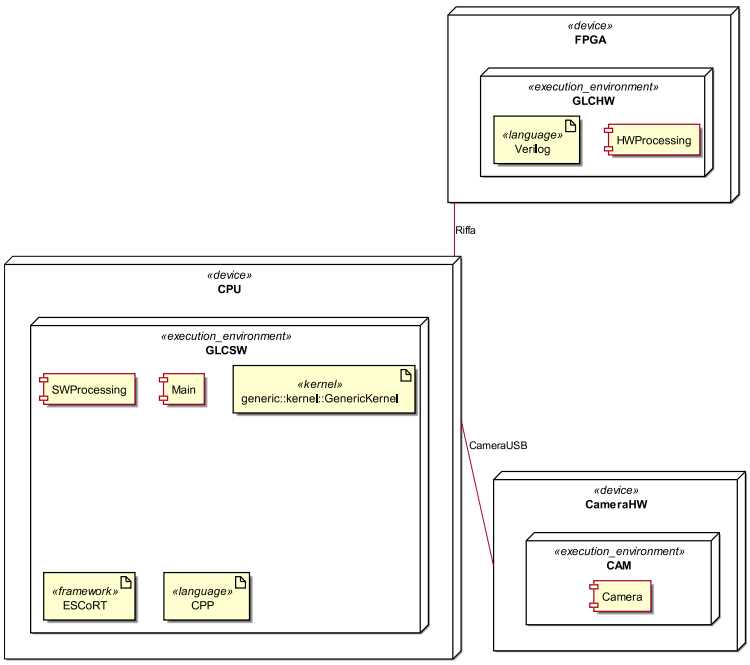


Figura 4.4: Diagrama de implantação do projeto Glaucoma

as transações e está fora do escopo deste trabalho. Já o diagrama de sequências determina as interações entre as instâncias de cada módulo. A Figura 4.3b especifica tais interações começando pelo evento inicial que chama o método *processFrame* do módulo *Main*. Por padrão, a execução das chamadas é feita de modo sequencial, assim, uma chamada síncrona ao módulo *Camera* é feita pelo pelo *processFrame*. Ao receber a imagem RGB, um bloco paralelo é executado. A extração dos dados da imagem é feita pelos módulos de *SWProcessing* e *HWProcessing*, inicialmente tem-se a extração dos limites do disco e da escavação e posteriormente tem-se o cálculo do diâmetro de ambos. Estas tarefas executam paralelamente, e, quando ambas finalizam, uma chamada ao serviço de calcular o VCDR do módulo de processamento de software é feita. O retorno é então salvo pelo módulo principal e o fluxo de execução finalizado para o evento executado.

A definição do hardware e seus parâmetros de configuração é definida no diagrama de implantação. Na Figura 4.4 é possível identificar que foram definidos três elementos, a CPU, FPGA e a câmera. Utilizando uma câmera USB, seu hardware não necessita ser implementado, somente um wrapper para fazer a chamada. O dispositivo FPGA irá conter o módulo de *HWProcessing* a ser implementado na linguagem Verilog. Neste projeto, não será gerado código nem implementado as estruturas de hardware. Por último, o dispositivo de CPU irá conter os módulos principal e de processamento de software. Será usado um kernel genérico na linguagem C++ e o framework ESCoRT. É possível também identificar nas conexões que o dispositivo FGPA se comunica com a CPU através da biblioteca Riffa, e com o hardware da câmera através de uma conexão USB.

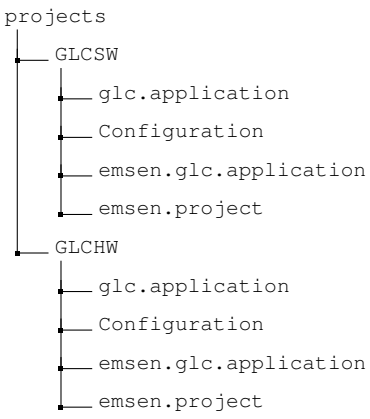


Figura 4.5: Estrutura de pastas do projeto Glaucoma

A Figura 4.5 apresenta a estrutura de pastas geradas. Foram gerados dois projetos, GLCSW e GLCHW, onde existem em ambos os módulos base no *namespace* "emsen.glc.application"e os módulos do usuário no *namespace* "glc.application". Como exemplo, a Figura 4.6 mostra uma possível implementação módulo da câmera, a área marcada foi implementado pelo projetista e consiste no recebimento do componente de driver da câmera USB, além de sua configuração e utilização durante a chamada ao serviço. De forma análoga, os módulos *SWProcessing* e *HWProcessing* possuem métodos a serem implementados referentes aos serviços que oferecem, cada um contendo quatro parâmetros, o contexto, os dados de entrada, um ponteiro para um objeto a ser retornado e o código do evento referente à chamada ao serviço.

Avançando no fluxo de execução, a Figura 4.7 ilustra a inicialização e a implementação de um contexto para o módulo *Main*. É possível observar que para cada chamada à serviço definida no diagrama de sequência, um método *request params* e *receive* são criados (exceto para o serviço *getFrame*, que só possui o método *receive*). As Figuras 4.8, 4.9 e 4.10 demonstram a implementação destes métodos. Em todas as figuras, a área marcada representa o trabalho feita pelo desenvolvedor. A implementação é semelhante em todos os casos e consiste em, no caso do *request params*, preencher a estrutura com os parâmetros do serviço, normalmente utilizando o contexto recebido e, nos estados de *receive*, ler o dado recebido e preencher no contexto para uso posterior.

```
CameraModule::CameraModule(
    USBCameraDriver *usbCameraDriver) :
    usbCameraDriver(usbCameraDriver)
{

}

void CameraModule::init()
{
    this->usbCameraDriver->open();
}

bool CameraModule::getFrame(
    IContext *context,
    EMSENCameraModule::DataType_getFrame_input input,
    RGBImage *output,
    unsigned int event)
{
    if (this->usbCameraDriver->isOpen())
    {
        return this->usbCameraDriver->getFrame(output);
    }
    else
    {
        return false;
    }
}
```

Figura 4.6: Implementação do módulo *Camera*

```
class Main {
private:
    class ProcessFrameContext : public IContext {
    public:
        RGBImage image;
        Limits discLimits;
        Limits cubLimits;
        unsigned int discDistance;
        unsigned int cubDistance;
        unsigned int VCDR;
    };
public:
    MainModule();
    virtual void init();
    virtual IContext* getExecutionContext();
protected:
    virtual void processFrame_Camera_getFrame_receive(
        IContext *context,
        EMSENCameraModule::DataType_getFrame_output output);

    virtual EMSENSWProcessingModule::DataType_extractDiscLimits_input
        processFrame_SWProcessing_extractDiscLimits_requestParams(IContext *context);
    virtual void processFrame_SWProcessing_extractDiscLimits_receive(
        IContext *context,
        EMSENSWProcessingModule::DataType_extractDiscLimits_output output);

    virtual EMSENSWProcessingModule::DataType_calculateDiscDistance_input
        processFrame_SWProcessing_calculateDiscDistance_requestParams(IContext *context);
    virtual void processFrame_SWProcessing_calculateDiscDistance_receive(
        IContext *context,
        EMSENSWProcessingModule::DataType_calculateDiscDistance_output output);

    virtual EMSENHWProcessingModule::DataType_extractCubLimits_input
        processFrame_HWProcessing_extractCubLimits_requestParams(IContext *context);
    virtual void processFrame_HWProcessing_extractCubLimits_receive(
        IContext *context,
        EMSENHWProcessingModule::DataType_extractCubLimits_output output);

    virtual EMSENHWProcessingModule::DataType_calculateCubDistance_input
        processFrame_HWProcessing_calculateCubDistance_requestParams(IContext *context);
    virtual void processFrame_HWProcessing_calculateCubDistance_receive(
        IContext *context,
        EMSENHWProcessingModule::DataType_calculateCubDistance_output output);

    virtual EMSENSWProcessingModule::DataType_calculateVCDR_input
        processFrame_SWProcessing_calculateVCDR_requestParams(IContext *context);
    virtual void processFrame_SWProcessing_calculateVCDR_receive(
        IContext *context,
        EMSENSWProcessingModule::DataType_calculateVCDR_output output);

private:
    ProcessFrameContext singleContext;
};

MainModule::MainModule(){}

void MainModule::init(){}

IContext* MainModule::getExecutionContext()
{
    return &this->singleContext;
}
```

Figura 4.7: Inicialização e contexto do módulo *Main*

```
EMSENSWProcessingModule::DataType_extractDiscLimits_input
MainModule::processFrame_SWProcessing_extractDiscLimits_requestParams(
    IContext *context)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    EMSENSWProcessingModule::DataType_extractDiscLimits_input input;

    input.image = pfcontext->image;

    return input;
}

void MainModule::processFrame_SWProcessing_extractDiscLimits_receive(
    IContext *context,
    Limits output)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    pfcontext->discLimits = output;
}

EMSENSWProcessingModule::DataType_calculateDiscDistance_input
MainModule::processFrame_SWProcessing_calculateDiscDistance_requestParams(
    IContext *context)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    EMSENSWProcessingModule::DataType_calculateDiscDistance_input input;

    input.limits = pfcontext->discLimits;

    return input;
}

void MainModule::processFrame_SWProcessing_calculateDiscDistance_receive(
    IContext *context,
    unsigned int output)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    pfcontext->discDistance = output;
}

EMSENSWProcessingModule::DataType_calculateVCDR_input
MainModule::processFrame_SWProcessing_calculateVCDR_requestParams(
    IContext *context)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    EMSENSWProcessingModule::DataType_calculateVCDR_input input;

    input.discDistance = pfcontext->discDistance;
    input.cupDistance = pfcontext->cupDistance;

    return input;
}

void MainModule::processFrame_SWProcessing_calculateVCDR_receive(
    IContext *context, unsigned int output)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    pfcontext->VCDR = output;
}
```

Figura 4.8: Estados *request params* e *receive* das chamada de serviço do módulo *Main* ao módulo *SWProcessing*

```
EMSENHWProcessingModule::DataType_extractCupLimits_input
MainModule::processFrame_HWProcessing_extractCupLimits_requestParams(
    IContext *context)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    EMSENHWProcessingModule::DataType_extractCupLimits_input input;

    input.image = pfcontext->image;

    return input;
}

void MainModule::processFrame_HWProcessing_extractCupLimits_receive(
    IContext *context,
    Limits output)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    pfcontext->cupLimits = output;
}

EMSENHWProcessingModule::DataType_calculateCupDistance_input
MainModule::processFrame_HWProcessing_calculateCupDistance_requestParams(
    IContext *context)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    EMSENHWProcessingModule::DataType_calculateCupDistance_input input;

    input.limits = pfcontext->cupLimits;

    return input;
}

void MainModule::processFrame_HWProcessing_calculateCupDistance_receive(
    IContext *context,
    unsigned int output)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    pfcontext->cupDistance = output;
}
```

Figura 4.9: Estados *request params* e *receive* das chamada de serviço do módulo *Main* ao módulo *HWProcessing*

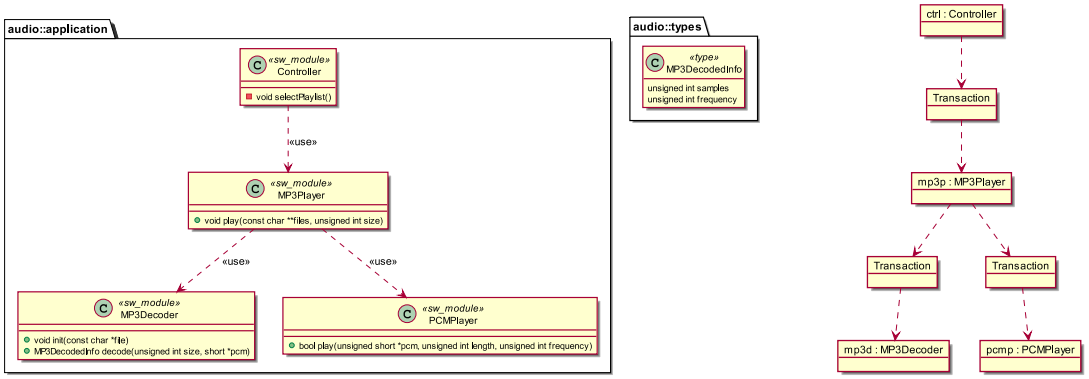
```
void MainModule::processFrame_Camera_getFrame_receive(
    IContext *context,
    RGBImage output)
{
    ProcessFrameContext* pfcontext = static_cast<ProcessFrameContext*>(context);

    pfcontext->image = output;
}
```

Figura 4.10: Estado *request params* da chamada de serviço *getFrame* do módulo *Main* ao módulo *Camera*

4.2 Player de áudio

O segundo projeto desenvolvido como estudo de caso foi um tocador MP3. Como é possível ver no diagrama de classes (Figura 4.11a), este projeto é composto por quatro módulos de software: o controlador (*Controller*), o tocador de playlists (*MP3Player*), o decodificador MP3 (*MP3Decoder*) e o tocador de áudio bruto (*PCMPlayer*). O controlador é responsável por selecionar uma playlist e enviá-la para ser tocada no tocador, por isso, o tocador oferece um serviço de tocar playlist que é usado pelo controlador. O tocador então utiliza os serviços oferecidos pelo decodificador MP3 (*init* e *decode*) para decodificar o MP3 em áudio bruto e o serviço de tocar áudio bruto oferecido pelo módulo *PCMPlayer*. Uma instância de cada módulo foi criada no diagrama de objetos (Figura 4.11b).

O ponto de entrada do tocador é o método *selectPlaylist* do controlador, este método permite que o controlador carregue uma playlist e chame o serviço play do *MP3Player* para tocá-la. Um *loop* então se inicia onde onde o tocador MP3 irá tocar cada áudio disponível na playlist. Para cada áudio, o decodificador MP3 é inicializado através da chamada ao serviço *init*, e então a primeira amostra é decodificada através do serviço *decode* de forma síncrona. Quando uma chamada deste tipo é feita, o fluxo de execução ficará bloqueado até que o serviço retorne uma resposta. Em seguida, um outro *loop* interno é inicializado para decodificar e tocar o áudio até que ele acabe. Neste *loop*, é inicialmente feita uma segunda chamada ao serviço de decodificação do decodificador MP3 de forma assíncrona, desta forma, é possível continuar o fluxo de execução até que a decodificação retorne um resultado. Enquanto a resposta do decodificador não retorna, a última decodificação válida é repassada ao serviço *play* do tocador de áudio bruto (*PCMPlayer*). Com isto, implementa-se um mecanismo de *double buffer* para que o áudio não possua interrupções devido a decodificação. Quando o *PCMPlayer* finalizar a execução do áudio bruto, é esperado que a decodificação MP3 já tenha sido finalizada e o loop possa executar novamente, porém, no caso da decodificação não ter sido finalizada, o loop irá esperar que a chamada assíncrona seja finalizada para ser executado. A Figura 4.12 ilustra todo este processo.



(a) Diagrama de classes (b) Diagrama de objetos

Figura 4.11: Modelagem do projeto de Áudio

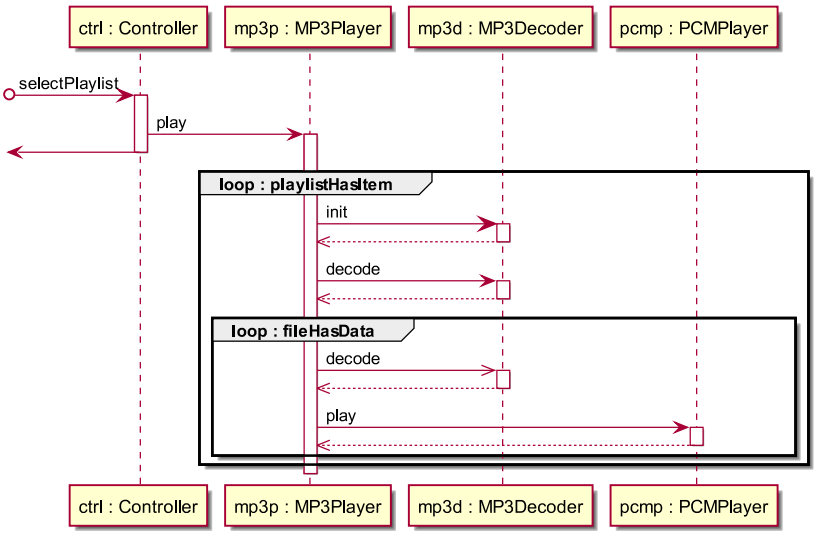


Figura 4.12: Diagrama de sequência do projeto Áudio

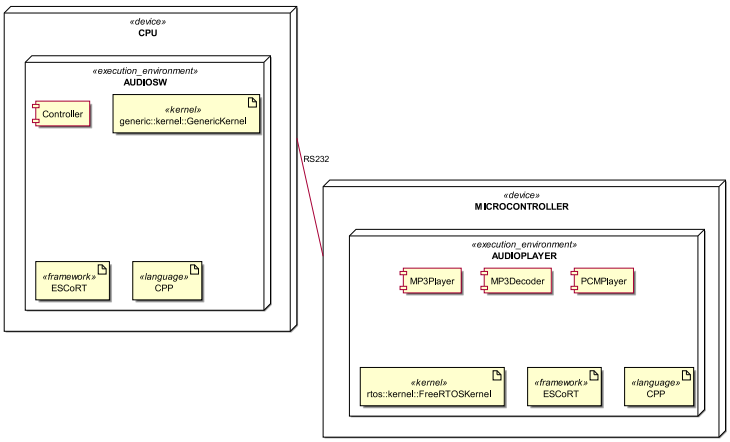


Figura 4.13: Diagrama de implantação do projeto Áudio

Este sistema de tocador MP3 foi desenvolvido para um microcontrolador ARM (Cortex-M3) utilizando o sistema operacional de tempo real *FreeRTOS*, o framework ESCoRT e a linguagem de programação C++. Os módulos de *MP3Player*, *MP3Decoder* e *PCMPlayer* foram alocados para o microcontrolador, enquanto o módulo *Controller* foi alocado em um processador de uso geral utilizando um kernel genérico, o framework ESCoRT e a linguagem C++. A Figura 4.13 ilustra a definição do hardware. A comunicação entre a CPU e o Microcontrolador foi feita através de uma interface RS232.

A estrutura geral de pastas e organização do projeto é semelhante ao diagnosticador de glaucoma. Uma diferença neste projeto, é a utilização de *loops*. A Figura 4.15 demonstra a implementação de ambos os *loops* do projeto, onde as áreas marcadas foram preenchidas pelo desenvolvedor. O primeiro verifica se a playlist ainda há itens a serem tocados, já a segunda

verifica se o arquivo atual ainda há dados a serem tocados. A utilização do contexto é essencial para troca de informações entre os diferentes estados do sistema. A execução dos *loops* é de responsabilidade do *EventDispatchQueue* que, ao receber um evento de *loop*, irá fazer a chamada ao método implementado pelo desenvolvedor e executará os subeventos repetidamente até que seja retornado um valor booleano falso.

O serviço *play* do módulo *MP3Player* é implementado na Figura 4.14, que também demonstra o objeto de contexto utilizado. Neste caso, a implementação do serviço é a inicialização do contexto que será utilizada durante o fluxo de execução. O contexto foi inicializado com dois buffers e os arquivos de playlist recebidos como parâmetros. Os métodos de *request params* e *receive* utilizados neste módulo são ilustrados na Figura 4.16. São feitas chamadas aos serviços *init* e *decode* do módulo *MP3Decode* e ao serviço *play* do módulo *PCMPlayer*.

```
class MP3PlayerModule {
private:
    struct PCMData
    {
        short pcm[1152];
        unsigned int samples;
        unsigned int frequency;
    } pcmdata[2];

    class MP3PlayerContext : Context
    {
        PCMData* current;
        PCMData* available;
        const char **files;
        unsigned int playlistSize;
        unsigned int currentFile;
    };

    [...]
};

IContext* MP3PlayerModule::getExecutionContext()
{
    return &this->singleContext;
}

bool MP3PlayerModule::play(
    IContext *context,
    EMSENMP3PlayerModule::DataType_play_input input,
    unsigned int event){
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    mp3pcontext->current = &this->pcmdata[0];
    mp3pcontext->available = &this->pcmdata[1];

    mp3pcontext->current->samples = mp3pcontext->available->samples = 0;

    mp3pcontext->files = input.files;
    mp3pcontext->playlistSize = input.size;
    mp3pcontext->currentFile = 0;

    return true;
}
```

Figura 4.14: Contexto e inicialização do serviço *play* do módulo *MP3Player*

```
bool MP3PlayerModule::loop_playlistHasItem(IContext* context)
{
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    return mp3context->currentFile < mp3context->playlistSize;
}

bool  MP3PlayerModule::loop_fileHasData(IContext* context)
{
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    return mp3pcontext->current->samples > 0 || mp3pcontext->available->samples > 0;
}
```

Figura 4.15: Métodos de *Loop* do módulo *MP3Player*

```
EMSENMP3DecoderModule::DataType_init_input
MP3Decoder::play_MP3Decoder_init_requestParams(
    IContext *context){
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    EMSENMP3PlayerModule::DataType_play_input input;

    input.file = mp3pcontext->files[mp3pcontext->currentFile++];

    return input;
}

EMSENMP3DecoderModule::DataType_decode_input
MP3Decoder::play_MP3Decoder_decode_requestParams(
    IContext *context){
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    EMSENMP3DecoderModule::DataType_decode_input input;

    input.size = 1152;
    input.pcm = mp3pcontext->available->pcm;

    return input;
}

void MP3Decoder::play_MP3Decoder_decode_receive(
    IContext *context,
    MP3DecodedInfo output){
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    mp3pcontext->available->availableSamples = output->samples;
    mp3pcontext->available->availableFrequency = output->frequency;

    if (mp3pcontext->current->samples == 0)
    {
        swap(mp3pcontext->current, mp3pcontext->available);
    }
}

EMSENPCMPlayerModule::DataType_play_input
MP3Decoder::play_PCMPlayer_play_requestParams(
    IContext *context){
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    EMSENPCMPlayerModule::DataType_play_input input;

    input.pcm = mp3pcontext->current->pcm;
    input.length = mp3pcontext->current->samples;
    input.frequency = mp3pcontext->current->frequency;

    return input;
}

void MP3Decoder::play_PCMPlayer_play_receive(
    IContext *context,
    bool output){
    MP3PlayerContext* mp3pcontext = static_cast<MP3PlayerContext*>(context);

    swap(mp3pcontext->current, mp3pcontext->available);
    mp3pcontext->available->samples = 0;
}
```

Figura 4.16: Métodos de *request params* e *receive* do módulo *MP3Player*

CAPÍTULO 5

Conclusão e trabalhos futuros

Neste trabalho, foi proposto uma infraestrutura de comunicação projetada para ser compatível com uma modelagem em nível de serviço. A infraestrutura proposta é independente das funcionalidades principais do sistema, possui fluxos de execução organizados com capacidade de quebra e reconstrução em dispositivos diferentes e suporte a paralelismos e chamadas síncronas ou assíncronas.

Como parte da modelagem de sistemas em nível de serviço, foi utilizado o profile UML-ESL. Foi então proposto o diagrama de implantação com o objetivo de estender sua funcionalidade para suportar a especificação de plataformas simplificadas. A partir desta especificação, foi proposto um gerador de código automático capaz de receber a especificação UML-ESL como entrada e prover um ou mais projetos em C++ como saída.

Em comparação com proposições anteriores, este trabalho oferece uma modelagem em nível de serviço em conjunto com uma especificação de plataforma simplificada. Dessa forma, há um menor esforço do projetista nas etapas iniciais do projeto, um menor acoplamento entre o modelo e plataformas específicas, além de clara separação entre as funcionalidades gerais do sistema das funcionalidades de comunicação. Utilizando a estratégia de componentes, a exploração de plataforma se torna possível com pouca ou nenhuma alteração nos modelos.

Novas pesquisas se fazem necessário para a evolução da modelagem utilizando UML-ESL e da infraestrutura proposta, o que inclui uma melhor especificação formal da semântica entre UML-ESL e os modelos de refinamento, proposição de infraestrutura em linguagem de hardware, geração de código para dispositivos de hardware, suporte a diferentes linguagens na geração de código de software, melhor tratamento de erros e alterações condicionais dos fluxos de execução.

Bibliografia

[1] M. de Andrade Almeida Gomes. "Sıntese de comunicação de sistemas modelados em nı-vel de serviços para plataformas baseadas em barramento". Diss. de mestr. Universidade Federal de Pernambuco, 2010.

[2] L. Cai e D. Gajski. "Transaction level modeling: an overview". Em: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2003, Newport Beach, CA, USA, October 1-3, 2003*. 2003, pp. 19–24.

[3] R. Carvalho, R. Alencar e A. Sarmento. "Generation of SystemC Simulation Models from Service Level UML Diagrams". Em: *VIII Brazilian Symposium on Computing Systems Engineering, SBESC 2018, Salvador, Brazil, November 5-8, 2018*. 2018, pp. 114–121.

[4] B. Councill e G. T. Heineman. "Definition of a software component and its elements". Em: 2001.

[5] P. C. Dantas, A. Sarmento e A. Sarmento. "A HW/SW embedded system for accelerating diagnosis of glaucoma from eye fundus images". Em: *2016 International Symposium on Rapid System Prototyping, RSP 2016, Pittsburg, PA, USA, October 6-7, 2016*. 2016, pp. 12–18.

[6] V. C. Dória. "Um ambiente para geração automática de biblioteca de componentes de comunicação em sistemas embarcados distribuıdos". Diss. de mestr. Universidade Federal de Pernambuco, 2003.

[7] H. Espinoza et al. "Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems". Em: *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*. 2009, pp. 98–113.

[8] D. D. Gajski et al. *Embedded System Design: Modeling, Synthesis and Verification*. 1st. Springer Publishing Company, Incorporated, 2009.

[9] V. Jain, A. Kumar e P. R. Panda. "A SysML Profile for Development and Early Validation of TLM 2.0 Models". Em: *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*. 2011, pp. 299–311.

[10] H. Kopetz. "The Complexity Challenge in Embedded System Design". Em: *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*. 2008, pp. 3–12.

[11] L. Lennis e J. Aedo. “Generation of Efficient Embedded C Code from UML / MARTE Models”. Em: 2013.

[12] MathWorks. *Simulink*. Disponível em: <https://www.mathworks.com/products/simulink.html>. Acesso em: 11 de nov. de 2019.

[13] G. A. F. B. de Melo. “ESCORT: um framework de componentes para a construção de software embarcado”. Diss. de mestr. Universidade Federal de Pernambuco, 2011.

[14] A. Nicolas et al. “Automatic deployment of component-based embedded systems from UML/MARTE models using MCAPI”. Em: *Design of Circuits and Integrated Systems*. Nov. de 2014, pp. 1–6.

[15] M. Nikolaidou et al. “Challenges in SysML Model Simulation”. Em: *Advances in Computer Science: an International Journal* 5 (jul. de 2016).

[16] S. Pasricha. “Transaction level modeling of SoC with SystemC 2.0”. Em: *Synopsys User Group Conference (SNUG)*. Vol. 3. 2002, p. 3.

[17] *PlantUML*. Disponível em: <https://plantuml.com>. Acesso em: 11 de nov. de 2019.

[18] E. Riccobene et al. “A model-driven design environment for embedded systems”. Em: *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*. 2006, pp. 915–918.

[19] R. V. de Sá Alencar. “Um Ambiente para desenvolvimento incremental de Sistemas Embarcados”. Trabalho de gradução. Universidade Federal de Pernambuco. 2016.

[20] D. C. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. Em: *Computer* 39.2 (fev. de 2006), pp. 25–31.

[21] ST. *STM32CubeMX*. Disponível em: <https://www.st.com/en/development-tools/stm32cubemx.html>. Acesso em: 11 de nov. de 2019.

[22] *Systems Modeling Language*. OMG. 2018.

[23] *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems*. OMG. 2008.

[24] H. Yu, R. Dömer e D. Gajski. “Embedded software generation from system level design languages”. Em: *Proceedings of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair 2004, Yokohama, Japan, January 27-30, 2004*. 2004, pp. 463–468.