Desenvolvimento dirigido a modelo para Bootloader de microcontrolador

André Luigi Bonote[1], Angeline Melchiors[1], Lucas Tosetto Teixeira[1] e Gian Ricardo Berkenbrock[1]

[1]Universidade Federal de Santa Catarina (UFSC)
Centro Tecnológico de Joinville
Caixa Postal 89.219-600 – Joinville – SC – Brazil

andreluigibonote@icloud.com, ange_melchiors@yahoo.com.br,

tosetto.lucas@gmail.com, gian.rb@ufsc.br

Abstract. *Embedded systems are used in several critical areas, and they are largely based on microcontrollers, so it is important to understand the use of tools that meet the requirements in code development for these systems. This paper brings a model-driven development approach to the ATmega328P microcontroller bootloader, to demonstrate the applicability of using automated code generation tools for this purpose. This was done by looking for a source code whose components can be reused by other applications. A UML model was generated in the Papyrus software, based on the source code provided for the Arduino Duemilanove, it was improved to the desired decoupling, and the components were used in an application developed to demonstrate their usability. Finally, the generated codes were subjected to static analysis and improved as demonstrated in the results.*

Resumo. *Os sistemas embarcados são utilizados em diversas áreas críticas e são na maioria baseados em microcontroladores. Por isso é importante compreender a utilização de ferramentas que atendam aos requisitos no desenvolvimento de código para estes sistemas. Este trabalho traz uma abordagem de desenvolvimento orientada por modelo para o bootloader do microcontrolador ATmega328P, para demonstrar a aplicabilidade do uso de ferramentas automatizadas de geração de código. Buscou-se utilizar um código-fonte cujos componentes possam ser reaproveitados por outras aplicações. Um modelo UML foi gerado no software Papyrus, baseado no código-fonte fornecido para o Arduino Duemilanove, foi aprimorado até atingir o desacoplamento desejado, então os componentes foram utilizados em uma aplicação desenvolvida para demonstrar sua usabilidade. Por fim, os códigos gerados foram submetidos à análise estática e aprimorados conforme apresentam os resultados.*

1. Introdução

Os microcontroladores estão presentes em significativa parte dos sistemas eletrônicos que utilizamos diariamente, e é um mercado que cresce continuamente [Network 2021]. Com isso surge a questão da atualização dos softwares, uma vez que existem milhões de unidades espalhadas pelo mundo [Beningo 2015]. Buscando

facilitar essas atualizações, este trabalho visa demonstrar a aplicabilidade da geração automática de códigos a partir de modelos *Unified Modeling Language* (UML) para criação de bootloaders, que consistem em programas instalados na memória dos microcontroladores com a capacidade de gravar e inicializar uma nova aplicação [Beningo 2015]. Espera-se que com essa geração automática facilite a implementação de recursos específicos nos códigos de bootloaders. Entretanto, nesse momento, deve ser validado se a geração automática está gerando códigos que respeitam as regras padrão, o que é abordado nesse trabalho.

Para fazer a modelagem e geração de códigos foi utilizado o software Papyrus. A verificação da qualidade dos códigos gerados é feita por análise estática, que verifica a adequação da conformidade do código com as regras padrão, utilizando as ferramentas Flawfinder e CPPCheck. Como padrão para este trabalho foi considerado o padrão de desenvolvimento MISRA C. O teste dos códigos gerados foi feito com microcontroladores Arduino Uno e Duemilanove para gravação do bootloader no ATMega328P e execução, respectivamente. Foi desenvolvido um circuito para apoiar a gravação e utilizada a IDE do Arduino, tanto para gravação do bootloader como da aplicação.

A seção 2 apresenta a definição dos principais conceitos necessários para entender os procedimentos realizados. Na seção 3 são descritos os procedimentos realizados, tanto para geração de códigos como análise e testes. Os resultados são apresentados na seção 4 e a seção 5 apresenta as conclusões em relação ao funcionamento e viabilidade dos procedimentos realizados.

2. Fundamentação

Nesta seção são apresentados os principais conceitos necessários ao entendimento do trabalho, partindo da visão de desenvolvimento dirigido a modelos, que compreende a automação de processos de transformação e geração de códigos a partir de modelos. Na sequência é explicada a técnica de análise estática, escolhida para avaliar a qualidade do código gerado conforme os padrões de programação e feita a definição do hardware utilizado para testes - os microcontroladores.

2.1. Desenvolvimento dirigido a modelos

O *Model Driven Development* (MDD) consiste em um processo de desenvolvimento de software guiado por modelos que descrevem o sistema [Sommerville 2019, Kleppe et al. 2003], buscando aumentar a produtividade, portabilidade e interoperabilidade entre plataformas, e facilitar a documentação e manutenção do software [Kleppe et al. 2003].

O ciclo de vida do MDD pode ser dividido em três etapas sequenciais automatizáveis com o uso de ferramentas: *Platform Independent Model* (PIM): modelo de alta abstração que busca descrever apenas o comportamento do software, independente da plataforma; *Platform Specific Model* (PSM): nível de abstração é menor, utiliza termos específicos da plataforma de implementação; código: tradução dos modelos em um código funcional. Um PIM pode gerar um ou mais PSMs e cada um destes gera um código. Ter mais de um PSM para um mesmo PIM aumenta a complexidade do projeto [Kleppe et al. 2003].

2.1.1. Geração de código

As ferramentas de geração de código fazem parte do processo de automatização na etapa de gerar um código funcional a partir do PSM, o que pode ser feito extraindo informações de um modelo por consultas e convertendo-as em fragmentos de texto baseados em regras previamente estabelecidas [omg 2008]. Usualmente, as ferramentas buscam por alvos que restringem os elementos do modelo para fazer a tradução. A eficácia advém da qualidade das buscas e dos algoritmos que interpretam como transcrever cada elemento do modelo [Tan et al. 2010].

A geração de código baseada em diagramas de classe costuma ser satisfatória, porém os comportamentos dos métodos presentes nas classes precisam ser feitos manualmente [Rumpe 2017], como, por exemplo, a inserção de comportamentos opacos, que são trechos de código na linguagem de programação que o diagrama representa no UML.

2.2. Análise estática do código

Para analisar a qualidade dos códigos gerados foi escolhido o método da análise estática. A análise é dita estática quando não ocorre em seu tempo de execução: apenas o texto do código é analisado em busca de erros, inconsistências e inseguranças. A análise estática detecta erros antes dos testes comumente empregados na indústria para sistemas embarcados, que em sua maioria são realizados tentando reproduzir a utilização do sistema dentro e fora do seu modo projetado de utilização [Reinbacher et al. 2009].

Por atuarem diretamente no código-fonte, as ferramentas de análise estática são rápidas, de baixo custo e simples por não precisarem de nenhuma sintaxe adicional. Podem não cobrir todas as classes de falhas, uma vez que identificam principalmente erros lógicos, e podem apresentar alertas falsos [Sommerville 2019].

2.2.1. Bootloaders

Bootloaders são programas cuja finalidade primeira é criar um mapa de memória do dispositivo onde se encontra e iniciar a aplicação [Uzlu and Saykol 2016]. Nos microcontroladores também realiza a gravação da aplicação na memória do dispositivo diretamente de um computador, sem a necessidade de equipamento externo, gerenciando a comunicação por um protocolo de comunicação serial [Mischie and Pazsitka 2019]. Usualmente, os bootloaders para MCUs são escritos para minimizar uso de memória. Precisam cumprir alguns requisitos para seu correto funcionamento, como habilidade de alternar entre o bootloader e a aplicação, possuir uma interface de comunicação e a capacidade de manipular as memórias de modo a gravar de forma correta a aplicação no microcontrolador [Beningo 2015].

O bootloader é executado quando o MCU é ligado. Por alguns instantes ele espera receber comandos pela comunicação serial, para realizar as ações para as quais foi escrito, como gravar informações na memória de programa ou apagar a memória de programa. Caso não receba nenhuma instrução, passa o controle da execução para a aplicação [Strickland 2016]. A diferença do bootloader para uma aplicação é o fato dele ser alocado em um espaço específico de memória que será executado ao ligar o MCU e

sua capacidade de manipular a memória de programa. A forma de escrever um bootloader é similar a de uma aplicação comum [Beningo 2015].

3. Materiais e Métodos

Nesta seção serão apresentadas as ferramentas e os métodos utilizados, incluindo o hardware, as técnicas de gravação na memória do microcontrolador, ferramentas para criação de modelos, geração automatizada de códigos e análise de código.

3.1. Hardware

As placas de Arduino Uno e Duemilanove foram escolhidas por serem de código aberto, permitindo a utilização e modificação dos bootloaders. Ambas utilizam o microcontrolador ATmega328P, que traz suporte nativo para bootloader. A gravação é possível através de uma interface de comunicação serial [atm 2015].

3.2. Gravação do bootloader

Para gravação do bootloader adicionou-se um soquete ZIF de 28 pinos diretamente no soquete original do MCU da placa Duemilanove, pela necessidade de retirar o MCU da placa original e colocá-lo na placa de gravação. A placa de gravação foi confeccionada manualmente conforme o esquemático da Figura 1. O circuito permite que seja usada uma placa de desenvolvimento Arduino, com o software de desenvolvimento fornecido pelo fabricante, para realizar a gravação do bootloader no ATMega328P [Arduino 2018].

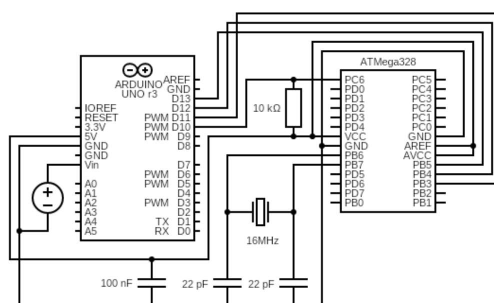

Figura 1. Circuito para gravar bootloader no ATmega328P com Arduino Uno

Fonte: o Autor (2021).

A gravação do bootloador foi feita com o arquivo “makefile” existente no diretório do Arduino IDE para compilar o código-fonte e gerar um arquivo “.hex” (hexadecimal). Caso se deseje um bootloader diferente da versão original é necessário mudar o código-fonte. A configuração do programador deve ser alterada para ”Arduino as ISP”na IDE para gravar corretamente o bootloader. Após esse processo o microcontrolador está pronto para receber uma aplicação.

3.3. Geração dos modelos e código

Para modelagem do bootloader usou-se o programa Papyrus, ferramenta de engenharia de código aberto para modelagem gráfica utilizando os conceitos de

UML [Papyrus 2020]. O programa permite geração de código com base nos modelos nas linguagens C, C++ e Java a partir do Papyrus Software Designer [Eclipse 2021].

Para geração de código em C funcional, foram adotadas as seguintes escolhas de modelagem: 1) Todas as classes são marcadas como não abstratas e recebem um estereótipo de um elemento da linguagem C, assim o gerador cria um arquivo de cabeçalho (".h") para cada classe, contendo uma estrutura (tipo *struct* do C) com os atributos e operações, caso existam. Quando existem operações na classe, é criado um arquivo ".c" com funções geradas a partir das operações. Assim é possível criar o encapsulamento desejado com o uso de arquivos. 2) Todos os elementos da classe (operações e atributos) foram colocados fora da estrutura, para que o comportamento estático do bootloader fosse melhor representado, bem como o acesso dos elementos fosse realizado por qualquer instância. 3) Todos os atributos são marcados como estáticos e únicos, e recebem um tipo compatível com a linguagem C, e o gerador os coloca fora da estrutura criada para a classe, garantindo seu caráter estático na memória do MCU. 4) Por padrão, ao utilizar estereótipos de C na classe, as operações são representadas como funções do C. Porém, assim como os atributos, todas as operações foram marcadas como estáticas e não consultivas. 5) Os métodos são descritos por comportamentos opacos, ativos, não recursivos e não abstratos, com a linguagem definida e o corpo escrito em C, garantindo o perfeito funcionamento das funções e permitindo que sejam reutilizados os códigos do bootloaders originais. Isso se deve principalmente às limitações do gerador de código, que não lida com modelos comportamentais na descrição de operações, apenas comportamentos opacos. 6) Toda associação é uma composição com multiplicidade [1..1], gerando a inclusão do cabeçalho da classe alvo na classe base.

3.4. Análise estática de software

Foi utilizado o padrão MISRA C para analisar os códigos gerados, pois este é o principal padrão de programação para desenvolvimento de sistemas embarcados. As ferramentas utilizadas foram o Flawfinder e o CPPCheck.

O Flawfinder busca por prováveis falhas de segurança em códigos escritos nas linguagens C e C++. Seu resultado elenca os erros em níveis que variam do 0, mais brando, ao 5, grande risco. As buscas são baseadas numa base de dados (BD) interna do programa. Ao encontrar um padrão entre o código e um erro do BD, ele acusará a possível falha [Flawfinder 2017]. O CPPCheck tem como característica importante a sua capacidade de definir facilmente um padrão de código para que a ferramenta analise se o código escrito é ou não compatível com o padrão. Diversos padrões são nativos, mas podem ser adicionados outros padrões com o uso de complementos, como o misra.py - um complemento que verifica a conformidade do código com o padrão MISRA C [cpp 2021].

4. Resultados

Os resultados, obtidos por meio do método exploratório, estão descritos nesta seção. Com base no modelo original fornecido pelo Arduino foram gerados sucessivos modelos, buscando, de maneira iterativa, aproximar-se do resultado desejado para o trabalho. A Figura 2 mostra todos os procedimentos e as versões de código realizadas até obter o código final.

As alterações em cada modelo deram-se nos seus aspectos estáticos, visando desacoplamento, permitindo a reutilização dos componentes do bootloader em outra

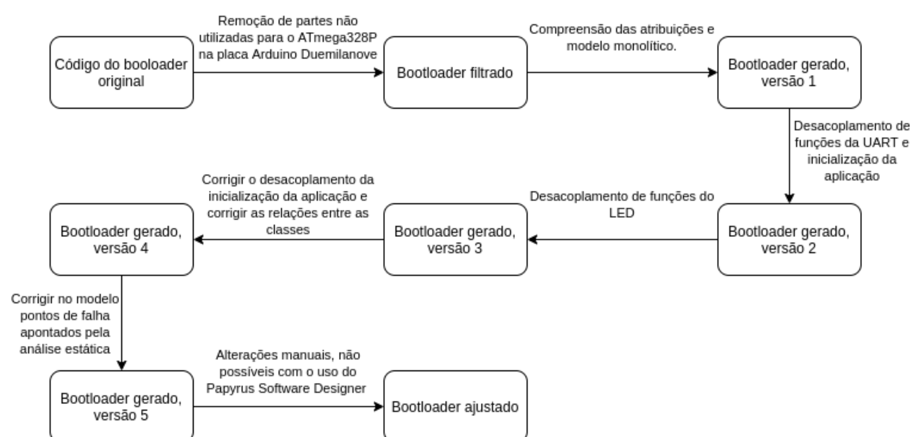

Figura 2. Diagrama de atividades dos códigos obtidos.

Fonte: o Autor (2021).

aplicação. Outro objetivo foi a melhoria da qualidade do código, principalmente segundo sua conformidade com o MISRA C. Testaram-se todas as versões de maneira prática por sua gravação no microcontrolador conforme descrito na Subseção 3.2 e seguida da gravação de uma aplicação de exemplo para verificar o funcionamento.

A existência de erros no funcionamento do bootloader não permitiria que a aplicação fosse gravada com sucesso. Todos os códigos de bootloader obtidos foram submetidos à análise estática, sendo que apenas o CPPCheck, quando utilizado com o complemento misra.py, demonstrou a presença de não conformidade com o MISRA C. O Flawfinder não apresentou possíveis falhas em nenhuma das análises.

4.1. Bootloader original

O bootloader fornecido para o Arduino Duemilanove é compatível com diversas placas da família Arduino, vários trechos do código servem apenas para lidar com essa pluralidade de aplicações. Como este trabalho explora apenas o uso da placa Duemilanove na versão que possui o MCU ATmega328P, a primeira etapa removeu manualmente os trechos de código desnecessários, facilitando a compreensão de seu funcionamento para realizar a engenharia reversa e determinar as atribuições do bootloader em questão.

Após as edições do código original, ele mantém as seguintes capacidades: realizar a chamada a aplicação, configurar e controlar o LED presente no Duemilanove, inicializar e realizar comunicação UART, e interpretar as mensagens enviadas pelo protocolo STK500, realizando as ações requeridas, como gravar a aplicação na memória do MCU.

4.2. Bootloader gerado, versão inicial

A partir do código original resultante da edição descrita na subseção 4.1 iniciou-se a modelagem, criando um modelo monolítico para manter a proximidade com o código original, que pode ser visto na Figura 3.

O único desacoplamento do modelo foi a separação dos tipos *union* e *struct*, uma vez que necessitam de estereótipo diferente do padrão de C utilizado para as demais classes. Foi criada uma classe que representa o código principal (ATmegaBOOT_168). As

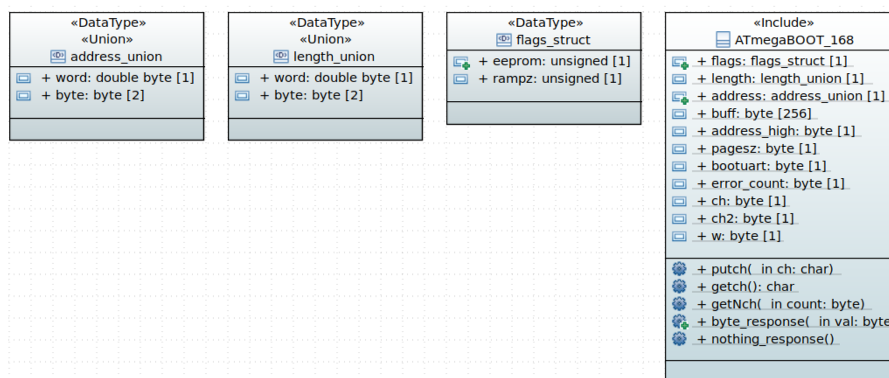

Figura 3. Primeira versão do modelo

Fonte: o Autor (2021).

uniões e estruturas foram adicionadas manualmente como atributos, referenciando o modelo de classe de cada uma como o tipo de atributo, bem como as outras variáveis do código. Foram adicionadas manualmente as operações na classe, tendo seu comportamento definido por comportamentos opacos retirados das funções do código original. No campo de corpo da classe ATmegaBOOT_168 acrescentou-se o ponteiro de função usado para iniciar a aplicação e a função *main*, definindo-a como a classe principal do novo bootloader. Ao gerar o código, criou-se um arquivo de cabeçalho para cada classe. Os cabeçalhos das uniões e estrutura são referenciados no cabeçalho da classe principal pelo gerador, por conta do uso dos modelos como tipos nos atributos da classe principal. Gerou-se, também, um arquivo de código contendo o comportamento das funções, a "main" e outros elementos de classe.

4.3. Considerações sobre análise estática

Para melhor analisar os dados das análises estáticas, são apresentados os gráficos de número de ocorrências de cada erro em cada versão do código (Figura 4) e do número de ocorrências totais de erros em cada versão do código (Figura 5).

Tendo como critério as diretrizes do padrão de código MISRA C, observa-se uma melhora na qualidade do código com o refinamento dos modelos. Comparando o primeiro código gerado com a versão proposta, apenas um tipo de erro teve um aumento de uma ocorrência, devido à decisão de melhorar o desacoplamento das atribuições do bootloader sem se distanciar muito do código original. Uma nova versão focada em corrigir aspectos comportamentais do modelo poderia eliminar esse erro, bem como reduzir ou eliminar outros erros restantes.

4.4. Aplicação

As atribuições de manipulação da UART e do LED possibilitaram o desenvolvimento de uma aplicação com o uso dos códigos gerados a partir das classes UART e Led visando mostrar a usabilidade desses códigos fora do contexto do bootloader. Seu modelo é observado na Figura 6.

A aplicação deve ser carregada no Arduíno pela IDE, que utiliza um arquivo principal com extensão ".ino". Isso implicou em uma série de diferenças entre o código para

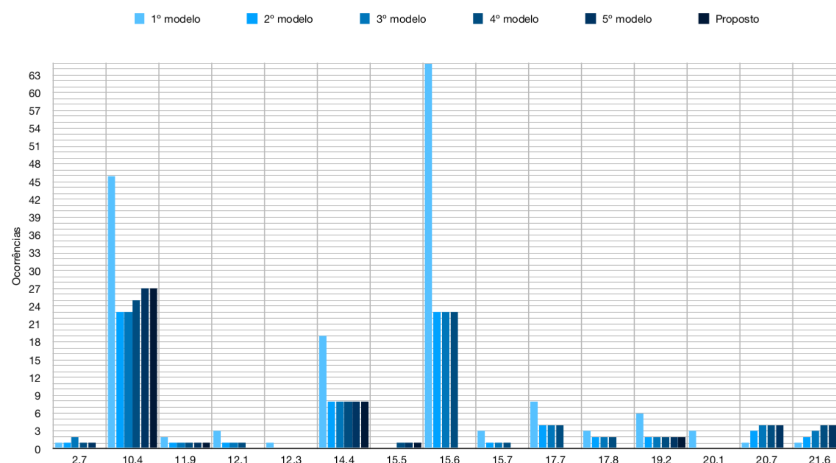

Figura 4. Ocorrências de cada erro por versão.

Fonte: o Autor (2021).

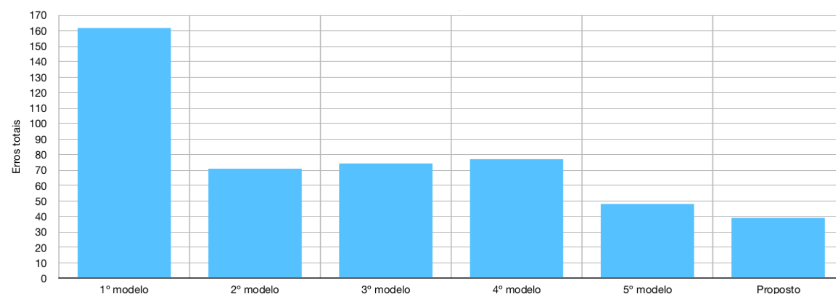

Figura 5. Ocorrências totais de erros por versão.

Fonte: o Autor (2021).

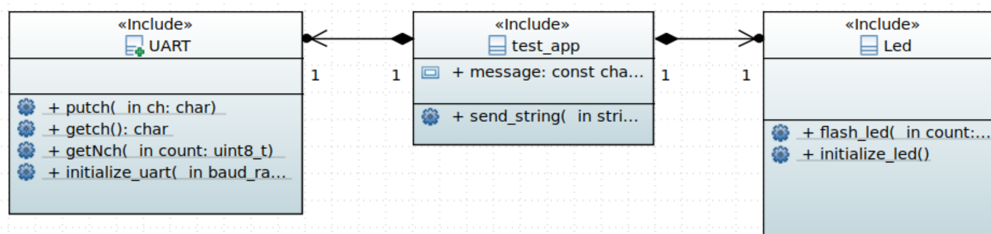

Figura 6. Modelo da aplicação de teste.

Fonte: o Autor (2021).

a IDE e um código, em C, exigindo alterações na maneira que o gerador constrói os códigos para ser possível utilizá-lo para gerar a aplicação para o Arduino. Em função disso, considerando também a simplicidade da aplicação, optou-se por escrever o código manualmente, obtendo o código apresentado na Figura 7.

Para utilização de Led e UART, mantiveram-se os arquivos ".c" e ".h" da versão desejada desses elementos, salvos no mesmo diretório o arquivo da aplicação (".ino"). Após o carregamento da aplicação no Arduino, observou-se seu correto funcionamento pelo comportamento do LED e também da mensagem recebida pelo monitor serial. Todas

```
extern "C" {
#include "Led.h"
#include "UART.h"
}

const char message[] = "HELLO";

void send_string(const char * string);

void setup() {
    initialize_led();
    initialize_uart(57600);
}
```

```
void loop(){
    send_string(message);
    delay(1000);
}

void send_string(const char * string){
    int str_length = strlen(string);
    flash_led(str_length);
    for (int i=0; i<str_length; i++){
        putch(string[i]);
    }
    putch('\n');
}
```

Figura 7. Código da aplicação.

Fonte: o Autor (2021).

as versões de Led e UART apresentaram compatibilidade com a aplicação, demonstrando o correto desacoplamento desses elementos do bootloader.

5. Conclusão

Este trabalho teve como objetivo apresentar e testar o uso de técnicas de desenvolvimento dirigido a modelos no desenvolvimento de bootloaders para microcontroladores.

A placa Arduino Duemilanove foi utilizada para os testes dos bootloaders gerados, sendo necessária uma placa Arduino Uno para a gravação de bootloader no ATmega328P. Após a gravação, uma aplicação de exemplo era salva diretamente na placa Duemilanove. Inicialmente foi alterado manualmente o código-fonte do bootloader para remoção das partes não relevantes para o estudo. Em seguida, foi desenhado no software Papyrus um modelo em UML representando o código obtido e com a ferramenta Papyrus Software Designer foi gerado um novo código e testado. Finalmente, analisou-se estaticamente o código com as ferramentas Flawfinder e CPPCheck.

As etapas utilizadas para a primeira versão modelada repetiram-se em outras duas versões, onde foi feito o desacoplamento das atribuições do bootloader, o controle da inicialização da aplicação, o controle da UART e do LED presente na placa. Na quarta versão foi melhorado o modelo e os aspectos que permitiram o uso dos componentes desacoplados em outras aplicações e passou-se a testar o desacoplamento por meio do uso de uma aplicação com esse fim. Na quinta versão foram realizadas alterações visando aumentar a conformidade do código com o padrão de código MISRA C.

A sexta versão foi obtida com edição manual da versão anterior. Não foi possível utilizar a geração automática, pois seriam necessárias alterações na maneira como a ferramenta - Papyrus Software Designer - traduz os modelos para código. Essa versão também foi submetida às análises estáticas e aos testes práticos.

Demonstrou-se, assim, a aplicabilidade de métodos de desenvolvimento dirigido a modelo para bootloader de microcontroladores, gerando-se um código-fonte com maior desacoplamento entre seus componentes, reutilizáveis em outra aplicação. Também se observou uma melhoria na qualidade do código, pelos parâmetros do padrão MISRA C, com pequenas alterações realizadas entre as versões dos modelos.

Referências

(2008). MOF - Model to Text Transformation Language.

(2015). Atmega328p datasheet. Rev.: 7810D–AVR–01/15.

(2021). Cppcheck manual. Ver. 2.5.

Arduino (2018). From arduino to a microcontroller on a breadboard. Disponível em: https://www.arduino.cc/en/Tutorial/BuiltInExamples/ArduinoToBreadboard. Acessado em 10 jun. 2020.

Beningo, J. (2015). Bootloader design for microcontrollers in embedded systems. *Embedded Software Design Techniques*.

Eclipse (2021). Papyrus software designer. Disponível em: https://marketplace.eclipse.org/content/papyrus-software-designer. Acessado em 20 ago. 2021.

Flawfinder (2017). Flawfinder. Disponível em: https://dwheeler.com/flawfinder/flawfinder.pdf. Acessado em 25 ago. 2021.

Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained*. Addison-Wesley Professional, Boston, MA, 1 edition.

Mischie, S. and Pazsitka, R. (2019). Designing a msp430 bootloader. In *In Proceedings of the 2019 INTERNATIONAL CONFERENCE ON APPLIED ELECTRONICS (AE)*. IEEE.

Network, E. N. (2021). Microcontroller market revenue to cross $20 billion by 2027: Global market insights. **Electronics B2B**. Disponível em: https://www.electronicsb2b.com/industry-buzz/microcontroller-market-revenue-to-cross-20-billion-by-2027-global-market-insights/. Acessado em 18 jul. 2020.

Papyrus (2020). Papyrus. Disponível em: https://www.eclipse.org/papyrus/. Acessado em 10 ago. 2021.

Reinbacher, T., Brauer, J., Horauer, M., and Schlicht, B. (2009). Refining assembly code static analysis for the intel mcs-51 microcontroller. In *In Proceedings of the 2009 IEEE INTERNATIONAL SYMPOSIUM ON INDUSTRIAL EMBEDDED SYSTEMS*.

Rumpe, B. (2017). *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer, New York, NY, 1 edition.

Sommerville, I. (2019). *Engenharia De Software*. Pearson Universidades, São Paulo, 10 edition.

Strickland, J. R. (2016). *Junk Box Arduino*. Springer Science+Business Media.

Tan, L., Yang, Z., and Xie, J. (2010). Ocl constraints automatic generation for uml class diagram. In *In Proceedings of the 2010 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND SERVICE SCIENCES*. IEEE.

Uzlu, T. and Saykol, E. (2016). Utilizing rust programming language for efi-based bootloader design. In *In Proceedings of the 2ND INTERNATIONAL CONFERENCE ON RECENT TRENDS AND APPLICATIONS IN COMPUTER SCIENCE AND INFORMATION TECHNOLOGY*. computer science bibliography.