Geração de Códigos usando Diagramas de Atividade para Sistemas Embarcados

Matheus Nunes Franco[1], **Lucas Eduardo Piana** [1], **Jean Marcelo Mira Junior** [1]
Gian Ricardo Berkenbrock[1],

[1]Laboratório de Sistemas Embarcados
Universidade Federal de Santa Catarina
Joinville, Santa Catarina, Brasil
mnunesfranco@gmail.com lucaspiana41@gmail.com
jean.mira@posgrad.ufsc.br gian.rb@ufsc.br

Abstract. *The development of embedded software in C++ is widely used in the industry, but it may require rework due to poorly defined requirements or communication issues. This work proposes an approach that uses UML behavioral activity diagrams to generate C++ code. Methods from the Debouncing pattern, applied to input devices such as buttons, were modeled and transformed into code by the tool. The automatic code generation proved to be efficient in object creation and manipulation, flow control with decision structures such as if and else, and function calls.*

Resumo. *O desenvolvimento de software embarcado em C++ é amplamente utilizado na indústria, mas pode exigir retrabalho devido a requisitos mal definidos ou falhas de comunicação. Este trabalho propõe uma abordagem que utiliza diagramas comportamentais de atividade UML para gerar código em C++. Métodos do padrão de Debouncing, aplicados a dispositivos de entrada como botões, foram modelados e transformados em código pela ferramenta. A geração automática de código mostrou-se eficiente na criação e manipulação de objetos, controle de fluxo com estruturas de decisão como if e else, e chamadas de funções.*

1. Introdução

Sistemas embarcados são amplamente utilizados na indústria para controlar aplicações predefinidas, executando um conjunto específico de tarefas. O software embarcado é responsável pelo controle do hardware, enquanto este último assegura o funcionamento adequado do sistema. O desenvolvimento de software embarcado exige planejamento rigoroso, para garantir que o hardware satisfaça as necessidades de memória e tempo de execução do sistema [White 2011].

O desenvolvimento de software é um processo demorado que requer análise cuidadosa e a utilização de linguagens de modelagem, como a *Unified Modeling Language* (UML), para atender aos requisitos do sistema e evitar falhas de comunicação. A UML é uma linguagem padronizada que facilita a especificação, visualização e documentação de sistemas complexos, promovendo uma comunicação eficaz entre a equipe e garantindo que os requisitos sejam compreendidos e atendidos.

Este trabalho explora o uso de C++ e UML no desenvolvimento de software para sistemas embarcados, com foco na transformação de modelos de diagramas de atividade

em código-fonte, visando criar soluções eficientes e claras, permitindo que a modelagem guie a geração automática de código, melhorando a precisão e a produtividade no desenvolvimento.

2. Fundamentação Teórica

Nesta seção, sera abordado o UML, com foco no Diagrama de Atividade, e a Transformação de Modelos em Código, explicando como a modelagem facilita a geração automática de código, suas vantagens e desafios.

2.1. Linguagem de Modelagem Unificada (UML)

A UML é uma linguagem visual amplamente utilizada para especificar, visualizar e documentar sistemas de software baseados em orientação a objetos [Guedes 2011][Silva and Videira 2001]. Ela é dividida em dois grandes grupos de diagramas, estruturais e comportamentais. Os diagramas estruturais definem a estrutura estática do sistema, enquanto os diagramas comportamentais modelam a transição dos estados do sistema ao longo do tempo [Group 2017].

Dentre os diagramas comportamentais, o diagrama de atividade será utilizado neste trabalho e seus aspectos serão demonstrados no próximo tópico.

2.1.1. Diagrama de Atividade

O diagrama de atividade é utilizado para especificar comportamentos em forma de atividades, com a possibilidade de adicionar parâmetros, pré-condições e pós-condições [Seidl et al. 2015]. A Figura 1 ilustra a modelagem de um diagrama de atividade no Eclipse Modeling. Todo conceito detalhado na Figura 1 é disponibilizado pela [Group 2017] na documentação da UML.

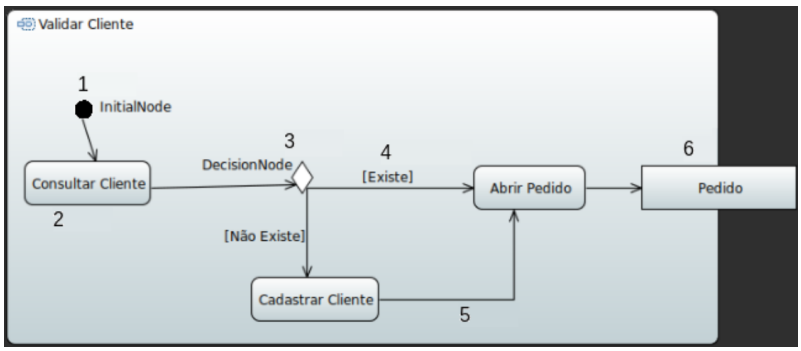

Figura 1. Modelagem do diagrama de atividade no Eclipse Modeling.

O nó 1 (*initialNode*) marca o início do fluxo de controle no diagrama de atividade. O nó representado pelo número 5 representa uma aresta que conecta e inicia um nó de atividade após o término do anterior. Posteriormente no número 2 (*opaqueAction)*contém uma estrutura de código predefinida, específica de implementação.

O nó 3 (*DecisionNode*) é um nó de controle que, com base em uma condição de guarda (número 4), determina o caminho do fluxo. A guarda só permite a transição do fluxo se a condição for verdadeira, geralmente expressa em termos booleanos. Por fim, o

nó representado pelo número 6 (*ActivityParameterNode*) contém objetos para entradas e saídas de atividades.

2.2. Transformação de Modelos em Código

O método para geração de código é uma técnica semelhante aos padrões de design orientado a objeto, pelo qual um programa tem capacidade e autonomia para gerar outro código, sendo o código um conjunto de palavras ou símbolos de forma ordenada, que transmite instruções de uma determinada linguagem de programação, ou seja, a geração de código tem como princípio criar programas que geram outros programas [Herrington 2003].

Uma das principais vantagens de desenvolver um gerador de código segundo [Herrington 2003] é poder controlar aspectos do desenvolvimento da ferramenta, bem como evitar erros gerados pela codificação manual. Deste modo [Herrington 2003] define como desvantagem do gerador de código o treinamento necessário para operar a ferramenta, tal como a necessidade de atualizar a ferramenta futuramente.

2.3. Metodologia

Os procedimentos necessários para a geração de código passam pela modelagem utilizando o software *Eclipse Modeling* [Project 2022] por meio da ferramenta *Papyrus* [Foundation 2022]. Na ferramenta *Papyrus* é realizada a modelagem do sistema. A abordagem teve início pela modelagem dos diagramas estruturais e, em seguida, dos diagramas comportamentais para cada método, por meio da criação de uma classe com seus respectivos atributos e métodos, sendo que nos métodos são empregados diagrama de atividade.

Após a modelagem, um arquivo .uml é gerado e interpretado pelo software Hugo-RT [Knapp 2022], uma ferramenta baseada em Java. O Hugo-RT gera código a partir de diagramas de classe UML. Este trabalho propõe uma abordagem para gerar código em C++ a partir de diagramas de atividades UML, na sequência apresentada pela Fig.2.

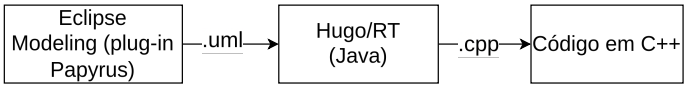

Figura 2. Abordagem Aplicada para Geração de Código.

2.3.1. Ambiente de Modelagem Eclipse Papyrus

O *Eclipse Papyrus* é uma ferramenta de modelagem independente que pode ser utilizada como um *plug-in* no software *Eclipse Modeling* [Project 2022]. Ele oferece suporte para *UML2*, conforme definido pela OMG, *SysML*, e sistemas em tempo real através da *UML-RT*. O Papyrus facilita a modelagem ao abstrair as características da *UML2*, auxiliando na configuração das propriedades e nas representações gráficas dos modelos, incluindo conjuntos de nós e arestas.

2.3.2. Modelagem

Primeiramente, é essencial definir a abordagem de modelagem e sua configuração para garantir uma modelagem coerente. Utilizando o *Eclipse Modeling* [Project 2022], cria-se uma estrutura de modelagem de classe com a ferramenta *Papyrus* chamado Registrador. Esta classe possui um atributo, endereço, que armazena um valor inteiro de 8 bits, e métodos como *setByte*, *getByte*, *clearByte* e *createObject*.

Cada método foi representado por um diagrama de atividade UML. Por exemplo, o diagrama do método *setByte* (ver Figura 3) inicia com um nó inicial (*InitialNode*) e segue por um controle de fluxo até um nó que lê a auto-ação (*ReadSelfAction*), acessando o objeto da classe em execução. O fluxo continua mediante um controle de objeto até um nó que adiciona um valor a um recurso estrutural (*AddStructuralFeatureValueAction*), recebendo-o por um nó de parâmetro de atividade (*ActivityParameterNode*) e finalizando no nó final (*ActivityFinalNode*).

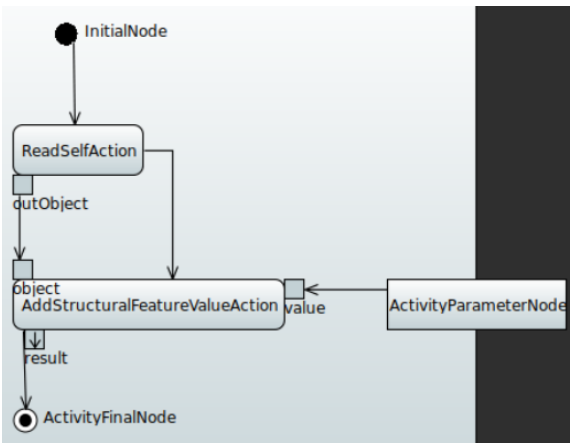

Figura 3. Diagrama de atividade setByte.

O diagrama de *getByte* (Figura 4) inicia com um nó inicial, segue para a leitura de um recurso estrutural (*ReadStructuralFeatureAction*), e termina com duas transições: uma para o nó final e outra para um nó de parâmetro de atividade, que gerencia os valores de entrada e saída.

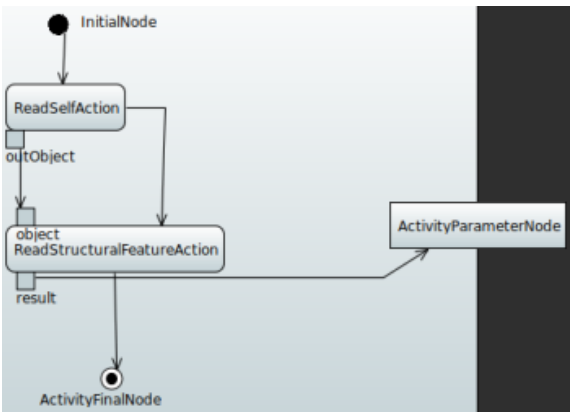

Figura 4. Diagrama de atividade getByte.

2.3.3. Hugo-RT

O Hugo-RT [Knapp 2022] é uma ferramenta projetada para gerar código e validar modelos UML, incluindo classes, máquinas de estado, interações e restrições OCL. A ferramenta gera código para sistemas verificadores de modelos em tempo real, como UPPAAL e SPIN, e também para linguagens de programação como Java e C++, especialmente para Arduino. A arquitetura do Hugo-RT é desenvolvida na linguagem Java.

De acordo com [Schäfer et al. 2001], o Hugo-RT é utilizado para a verificação de máquinas de estado cronometradas com modelagem UML. A configuração ativa de uma máquina de estados e o conteúdo da fila de eventos determinam o estado atual. A fila de eventos contém eventos não tratados, e os estados ativos formam uma árvore de estados.

2.3.4. Método

Após modelar os diagramas de atividade da classe, a ferramenta Hugo-RT [Knapp 2022] é compilada com o diretório do arquivo .uml. Em seguida, o arquivo .uml é lido, começando pela busca do nó inicial. Um laço *while* é executado até encontrar o nó de atividade final. Durante esse processo, é gerado código para cada nó de atividade, seguindo o fluxo de controle definido pela modelagem. Cada diagrama de atividade possui um nó inicial e um nó final.

Se houver um nó de decisão, ele deve ser empilhado em uma variável local. Todos os nós de decisão são empilhados sucessivamente. A sequência dos nós de controle segue suas dependências até encontrar um nó de junção. Quando um nó de junção é encontrado, o nó de decisão no topo da pilha é removido, e a sequência de controle é ajustada para que as duas arestas do nó de decisão tenham seus próximos nós verificados e o código seja gerado.

Cada nó de atividade proveniente da modelagem tem uma estrutura de programação específica. A abordagem utilizada acessa características específicas, baseando-se em um diagrama de classe retirado do livro de padrões de projeto para sistemas embarcados em C [Douglass 2011].

3. Resultados e Discussões

Uma abordagem de modelagem para solução do padrão Debouncing retirado do livro de padrões de projeto para sistemas embarcados em C [Douglass 2011], e modelado por meio da ferramenta Eclipse Papyrus Modeling é apresentada na Figura 5. Este padrão é usado em dispositivos de entrada de sistemas digitais, como botões e relés, onde a deformação do metal pode causar erros devido à alta velocidade de resposta do sistema. Para minimizar esses sinais, é necessário esperar um período após o acionamento do dispositivo para verificar o sinal gerado.

Foram implementados dois métodos, o método *eventReceive* da classe *ButtonDriver*, Figura 6 e o método *delay* da classe *Timer*, Figura 7.

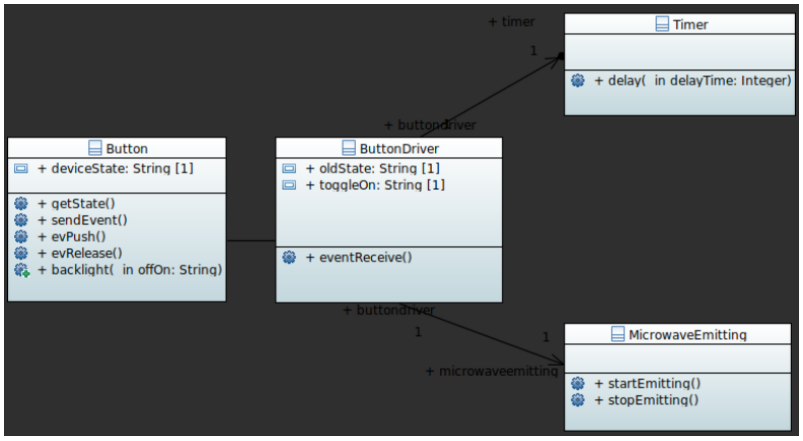

Figura 5. Diagrama de classe ButtonDriver com Eclipse Papyrus Modeling.

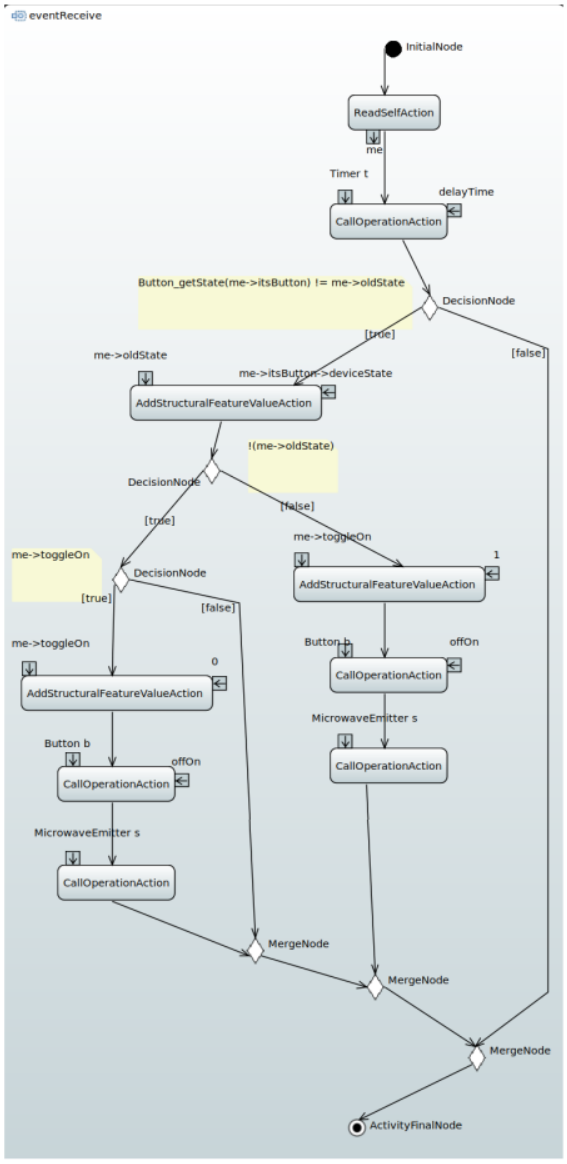

Figura 6. Diagrama de atividade eventReceive da classe ButtonDriver com Eclipse Papyrus Modeling.

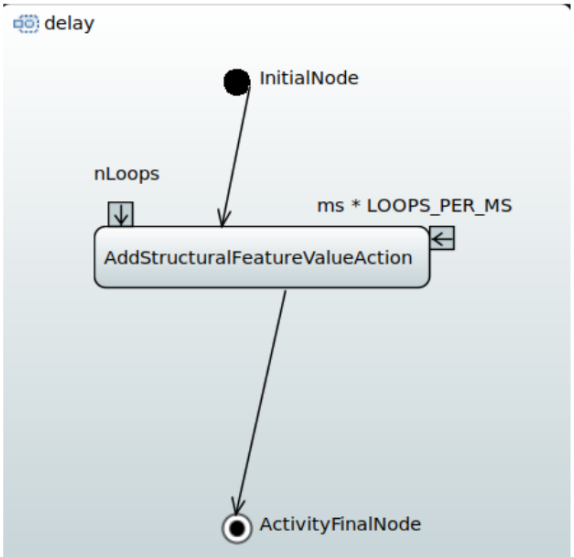

Figura 7. Diagrama de atividade delay da classe Timer com Eclipse Papyrus Modeling.

O padrão disposto por [Douglass 2011] para a solução em código para o método *eventReceive* pode ser visualizado na Figura 8.

```
void ButtonDriver_eventReceive(ButtonDriver* const me) {
        Timer_delay(me->itsTimer, DEBOUNCE_TIME);
        if (Button_getState(me->itsButton) != me->oldState) {
            /* must be a valid button event */
            me->oldState = me->itsButton->deviceState;
            if (!me->oldState) {
                /* must be a button release, so update toggle value */
                if (me->toggleOn) {
                    me->toggleOn = 0; /* toggle it off */
                    Button_backlight(me->itsButton, 0);
                    MicrowaveEmitter_stopEmitting(me->itsMicrowaveEmitter);
                    }
                else {
                    me->toggleOn = 1; /* toggle it on */
                    Button_backlight(me->itsButton, 1);
                    MicrowaveEmitter_startEmitting(me->itsMicrowaveEmitter);
                    }
                }

            /* if it's not a button release, then it must
               be a button push, which we ignore.
            */
        }
}
```

Figura 8. Código idealizado para solução do método *eventReceive* da classe ButtonDriver.

E a solução de (DOUGLASS, 2011) para o método delay é o código disposto na Figura 9.

```
/* LOOPS_PER_MS is the # of loops in the delay() function
   required to hit 1 ms, so it is processor and
   compiler-dependent
*/
#define LOOPS_PER_MS 1000
void delay(unsigned int ms) {
    long nLoops = ms * LOOPS_PER_MS;
    do {
    while (nLoops-);
    }
```

Figura 9. Código idealizado para solução do método *delay* da classe *Timer*.

A solução proposta para esses métodos são geradas por meio dos diagramas de atividades correspondentes a cada um, sendo que o código gerado do método *eventReceive* pode ser observado na Figura 10. E a Figura 11 corresponde ao código gerado para o método *delay*.

```
void ButtonDriver::eventReceive() {
  auto me = this;
  Timer t.delay();
  if (Button.getState(me->itsButton) != me->oldState) {
  me->oldState = me->itsButton->deviceState;
  if (!(me->oldState)) {
  if (me->toggleOn) {
  me->toggleOn = 0;
  Button b.backlight();
  MicrowaveEmitter s.startEmitting();

  }
  } else {
  me->toggleOn = 1;
  Button b.backlight();
  MicrowaveEmitter s.stopEmitting();

  }

  }
}
```

Figura 10. Código gerado para método *eventReceive* da classe *ButtonDriver*.

```
#include "../include/Timer.hh"

Timer::Timer() {
}

void Timer::setButtondriver(ButtonDriver* buttondriver) {
  this->buttondriver = buttondriver;
}

ButtonDriver* Timer::getButtondriver() {
  return this->buttondriver;
}

void Timer::delay(int delayTime) {
  nLoops = ms * LOOPS_PER_MS;
}
```

Figura 11. Código gerado para método *delay* da classe *Timer*.

3.1. Limitações

A implementação da geração de código com a ferramenta Hugo-RT apresentou várias limitações, como dificuldades com laços de repetição, falta de suporte para bibliotecas externas e novos tipos primitivos, além de problemas de indentação. A ferramenta também tem restrições ao gerar código para diagramas de atividades e não lida bem com certos

operadores e palavras-chave, resultando em erros. Modelos complexos, como os que envolvem tipos primitivos como *double*, enfrentam problemas de compilação. A Hugo-RT também apresenta dificuldades na especificação de comportamentos condicionais e na semântica de equações. Apesar de economizar tempo, a ferramenta ainda necessita de melhorias para oferecer maior flexibilidade e precisão.

3.2. Discussões

Certas características contribuem para uma modelagem mais fluida e facilitam o processo de geração de código. A forma literal como a UML especifica suas ações e como a linguagem de programação Java define suas funcionalidade colaboram para esse fato. A ferramenta *Eclipse Papyrus Modeling* colabora para obter um panorama das propriedades bem como facilitar o processo de criação das modelagens. Quando a modelagem se torna mais complexa o *Eclipse Papyrus Modeling* apresenta alguns erros de atualização da modelagem que esta sendo criada, sendo necessário atualizar a aba na ferramenta. Alguns nós apresentam falhas nas características, como o nó inicial com um círculo de cor branca.

A ferramenta Hugo-RT automatiza a geração de código dos métodos essenciais, economizando tempo. Contudo, a nomeação automática dos métodos pode ser pouco intuitiva, como *set0* ou *get1*, o que pode levar a métodos com nomes não convencionais.

Em algumas ações, é possível especificar mais informações, o que permite gerar equações em código sem explicitá-las diretamente nos pinos de valor ou entrada. No entanto, como nem todas as ações possuem as mesmas características, foi decidido padronizar as instruções nos pinos de entrada e valor. Por exemplo, a ação *StructuralFeatureValueAction* detalha o tipo do objeto de entrada conforme um atributo da classe, mas não permite definir um tipo primitivo, como ocorre com a ação *ActivityParameterNode*. Já a ação *ReadSelfAction* não permite especificar um tipo. Essas limitações refletem o papel de cada ação no arranjo da modelagem, justificando a padronização das instruções nos pinos de entrada, saída e valor.

Outro aspecto importante é uma falha na ferramenta *Eclipse Papyrus Modeling* ao exibir comentários. A ferramenta apresenta um bloco de comentário com fundo branco, mesmo com texto presente. Para corrigir esse erro, é necessário dar um duplo clique no comentário para que o texto apareça corretamente.

4. Conclusão

O projeto e desenvolvimento de software embarcado realizado por meio da abordagem dirigida a modelos tem como objetivo capturar os requisitos envolvidos no projeto e projetar sua arquitetura, evitando ambiguidades. A modelagem minimiza o retrabalho causado por falhas de comunicação no projeto, permitindo que alterações na modelagem resultem em mudanças no código gerado.

Neste trabalho, utilizou-se a Linguagem de Modelagem Unificada (UML) dentro da abordagem dirigida por modelos para modelar diagramas comportamentais de atividade e diagramas estruturais de classe, com foco no desenvolvimento de software para sistemas embarcados. Exemplos concretos, como os métodos *eventReceive* e *delay* do padrão de *Deboucing*, foram implementados e comparados com as soluções propostas

na literatura, revelando que a abordagem automatizada pode alinhar-se bem com padrões estabelecidos.

Conseguindo com essas ações e nós suportados na ferramenta Hugo-RT realizar a geração de código em C++ para criação de objetos, destruição de objetos, atribuir valor aos objetos, retornar o valor dos objetos, criar instruções de escolha para a estrutura por meio do *if* e *else*, realizar a chamada de funções para os objetos. Tendo como limitação os laços de repetição *while* e *for*, que colaboram para obter trechos de código incompletos. Os trabalhos futuros podem melhorar a geração de códigos com suporte de uma quantidade maior de ações comportamentais. Por exemplo, adicionando os laços de repetição na leitura da ferramenta Hugo-RT, essa adição pode ser feita por meio do nó *LoopNode* contido nos diagramas comportamentais de atividade. Além de aperfeiçoar os algoritmos de geração, trabalhos futuros poderão melhorar os algoritmos de geração de código com o suporte a um maior número de ações comportamentais, como a adição de laços de repetição por meio do nó *LoopNode* nos diagramas comportamentais de atividade. Além disso, será possível aperfeiçoar os algoritmos de geração de código, aumentando as ações e nós suportados pela ferramenta Hugo-RT. Por fim pode-se melhorar aspectos referentes a visualização do código produzido.

Referências

Douglass, B. P. (2011). *Design Patterns for Embedded Systems in C: An Embedded Software*. Elsevier, Oxford.

Foundation, T. E. (2022). Eclipse papyrus modeling environment. Accessed: 2022-05-31.

Group, O. M. (2017). Unified modeling language (uml) specification. In Group, O. M., editor, *UML Specification Documentation*, pages 1–500.

Guedes, G. T. A. (2011). *UML 2: Uma Abordagem Prática*. Novatec Editora, São Paulo, 2nd edition.

Herrington, J. (2003). *Code Generation in Action*. Manning Publications, Massachusetts.

Knapp, A. (2022). Hugo/rt. Accessed: 2022-05-31.

Project, E. M. (2022). Eclipse ide. Accessed: 2022-05-12.

Schäfer, T., Knapp, A., and Merz, S. (2001). Model checking uml state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357–369. Workshop on Software Model Checking (in connection with CAV '01).

Seidl, M. et al. (2015). *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer, Heidelberg.

Silva, A. M. R. and Videira, C. A. E. (2001). *UML: Metodologias e Ferramentas CASE*. Centro Atlântico, Lisboa.

White, E. (2011). *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media.