UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

JEAN MARCELO MIRA JUNIOR

GERAÇÃO DE CÓDIGO USANDO DIAGRAMAS DE ATIVIDADE PARA SISTEMAS EMBARCADOS

Joinville
2022

JEAN MARCELO MIRA JUNIOR

GERAÇÃO DE CÓDIGO USANDO DIAGRAMAS DE ATIVIDADE PARA SISTEMAS EMBARCADOS

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica no curso de Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville.

Orientador:Prof. Dr. Gian Ricardo Berkenbrock

Joinville
2022

Dedico esse trabalho ao meu pai Jean Marcelo Mira e à minha mãe Telma Daniela Gruner Mira, por sempre me instigarem a buscar conhecimento.

AGRADECIMENTOS

Aos Meus pais por sempre me incentivarem no decorrer de minha vida acadêmica.
À toda minha família que sempre me apoiou no transcorrer da graduação, em especial aos meus avós e minha madrinha.
Aos meus amigos por dividirem conhecimento e pelas conversas inspiradoras.
Ao meu orientador Prof. Dr. Gian Ricardo Berkenbrock pelo conhecimento transmitido.
À Universidade Federal de Santa Catarina e todos que contribuíram para este trabalho.

The function of good software is to make the complex appear to be simple.(BOOCH, 2010).

RESUMO

O processo de desenvolvimento de software embarcado na linguagem de programação C++ com aplicação em sistemas embarcados é amplamente difundida na indústria. Esse processo pode demandar retrabalho se não desenvolvido com requisitos de projeto bem estabelecidos, ou por falha de comunicação no decorrer do desenvolvimento do *software*. Consequentemente efetuar a alteração de valores de atributos ou métodos em um código com abundância de linhas representa uma perda desnecessária de recursos de uma empresa. Ocasionando no desenvolvimento de ferramentas para geração de código, criadas por empresas de *software*, visando facilitar futuras alterações na estruturas dos *softwares* projetados. Esse trabalho apresenta uma abordagem baseada em diagrama comportamentais de atividade UML para geração de código na linguagem de programação C++, portanto, fazendo a transformação de modelo para texto. Realizando a transformação de modelos de diagrama de atividade, por intermédio da linguagem de programação Java, resultando em um código na linguagem de programação C++.

Palavras-chave: Transformação. Diagramas comportamentais. UML. Geração de código. C++.

ABSTRACT

Developing embedded software in the C++ programming language applied to embedded systems is widespread in the industry. This process can require rework if not developed with well-established project requirements, or due to miscommunication during software development. Consequently, changing attributes or method values in a code with an abundance of lines represents a company's unnecessary loss of resources. This leads to the development of code generation tools, created by software companies, to facilitate future changes in the structure of the designed software. This work presents an approach based on UML behavioral activity diagrams for code generation in the C++ programming language, thus making the transformation of a model into text. The transformation of activity diagram models, by means of the Java programming language, results in a C++ programming language code. The code generated was satisfactory, with some drawbacks or limitations.

Keywords: Transformation. Behavioral diagrams. UML. Code generation. C++.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas embarcados são amplamente utilizados em projetos na indústria, com intuito de controlar uma aplicação predefinida de modo a executar um conjunto de tarefas. *Software* embarcado realiza o controle do hardware, ou e hardware no que lhe concerne realiza o controle do sistema em questão. O projeto de software embarcado necessita de planejamento para que o hardware comporte a necessidades de memória e tempo de execução (WHITE, 2011).

O desenvolvimento de um projeto de software embarcado é realizado para controlar e fornecer instruções ao *hardware*, sendo o software uma ferramenta capaz de otimizar e automatizar processos. Durante o processo de elaboração de um software as empresas buscam atender requisitos estipulados para o determinado sistema embarcado. Sendo assim, as empresas utilizam linguagem de modelagem para transmitir uma ideia, abordagem que se faz necessária pelas diversas possibilidades de falha na comunicação durante um projeto (WHITE, 2011).

O processo de desenvolvimento de um software tende a ser demorado, visto que requer análise do que se deseja obter, instituições utilizam os diagramas padronizados UML, sigla de Unified Modeling Languade (UML) ou em português, linguagem de modelagem unificada. A utilização da UML facilita a comunicação entre as pessoas envolvidas no projeto, pois elucida a visão do projeto através de diagramas padronizados (GROUP, 2005). No fim do processo, tendo-se a concepção do diagrama UML, a equipe responsável pelo esboço deverá se dedicar por mais um tempo. Para o desenvolvimento do código.

Os diagramas UML podem ser separados em dois grandes grupos, diagramas estruturais e diagramas comportamentais. O grupo dos diagramas estruturais é caracterizado por abstrair a forma estática de um sistema, contendo nele os digramas de classe, componentes, implementação, objetos e pacotes. Já o grupo dos diagramas comportamentais é caracterizado por abstrair a forma dinâmica de um sistema, contendo nele os diagramas de atividade, comunicação, sequência, máquina de estados, tempo e caso de uso (OMG, 2017).

Visando o aprimoramento do desenvolvimento de projetos de sistemas embarcados, este trabalho apresenta uma análise de geração de código, focando no desenvolvimento e aplicação do grupo comportamental dos diagramas UML, ou seja, especificamente os diagramas UML de atividade. A pesquisa ora apresentada visou diminuir o tempo entre o desenvolvimento e concepção de um programa através de um método que consiga captar as essência do diagrama UML de atividade e a partir disso gerar códigos na linguagem C++.

1.1 OBJETIVO

Para resolver a problemática da geração de código na linguagem C++ através de uma abordagem de modelagem, propõe-se neste trabalho os seguintes objetivos:

1.1.1 Objetivo Geral

Elaborar um procedimento para geração de código utilizando modelos UML voltado para diagramas comportamentais, especificamente o diagrama de atividades.

1.1.2 Objetivos Específicos

- Estudar a literatura sobre transformação de modelo para texto;
- Selecionar uma abordagem para geração de código através de diagramas comportamentais UML;
- Projetar e implementar a geração de código;
- Testar e analisar os resultados da implementação.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda, teoricamente, os tópicos necessários para a concepção de projeto, análise e proposta de solução, assim, os conceitos demonstrados partem dos elementos de sistemas embarcados, da criação do projeto através de diagramas UML, da análise por meio desenvolvimento dirigido por modelos e da concepção através de um algoritmo que permita a geração do código.

2.1 SISTEMAS EMBARCADOS

Pode-se definir um sistema embarcado como "[...] um sistema computadorizado dedicado a executar um conjunto específico de funções reais, em vez de fornecer um ambiente de computação generalizado."(DOUGLASS, 2011, p. 1). Ainda conforme White (2011), uma maneira de definir um sistema embarcado sem discutir as tecnologias empregadas do mesmo, é expressar tal como um sistema computadorizado que tem como princípio, na sua construção, a solução de uma aplicação, além de ser limitado em comparação a computadores de uso geral.

Sistemas embarcados contam com limitações de hardware como, uma CPU que funcione mais lentamente para economizar bateria. Ou um sistema que use menor quantidade de memória para reduzir o custo sobre o produto. Como restrições no desenvolvimento de software impostos pela problemática que se busca resolver. Como exemplo de restrições no desenvolvimento têm-se softwares que devem executar de maneira exata cada ciclo (deterministicamente), sistemas de tempo real que necessitam de um software que responda de maneira rápida a determinado evento, sistemas que necessitam de tolerância a falhas para executar, mesmo que mensagens de erros decorrentes da degradação normal ocorram ou sistemas que exigem clareza nas mensagens de erro (WHITE, 2011).

Desse modo, Douglass (2011) define como requisitos para um sistema embarcado, ter confiabilidade, robustez e segurança, onde a confiabilidade é uma medida estocástica de probabilidade que quantifica a possibilidade do correto funcionamento dos processos, a robustez descreve a capacidade de um sistema de prover serviços satisfatoriamente quando suas pré-condições são transgredidas e a segurança refere-se à probabilidade de que a execução do sistema provenha de uma casualidade ou perda, ou seja, indica o nível de risco de um sistema.

Ünsalan, Gürhan e Yücel (2022), descrevem um sistema embarcado como um sistema que funciona autonomamente na maioria das vezes podendo se comunicar com a internet das coisas (IoT - Internet Of Things). Existe uma diversidade de sistemas embarcados disponíveis no mercado, aplicados em áreas diversificadas

como eletrodoméstico, sistemas bélicos, sistemas automotivos, sistemas hospitalares e sistemas de controle voltados para a indústria. Sendo assim, Ünsalan, Gürhan e Yücel (2022) define um sistema embarcado como um sistema que utiliza hardware e software para efetuar a aquisição de dados, processamento e produza um resultado coerente com sua aplicabilidade. Ünsalan, Gürhan e Yücel (2022) especifica que a utilização de microcontroladores baseados em Arduino e ARM Cortex-M é benéfico para aplicação em sistemas embarcados pela ampla área de uso e aplicação, mesmo que possua recursos limitados.

2.2 DESENVOLVIMENTO DIRIGIDO POR MODELOS

Durante o desenvolvimento de software, segundo Kleppe, Warmer e Bast (2005), há inicialmente a produção dos conceitos necessários para o projeto proposto seguido pelo levantamento de requisitos, análise e descrição funcional, elaboração do projeto, codificação e, por fim, o teste do código desenvolvido, além dos ajustes necessários. Essa abordagem nas fases iniciais de desenvolvimento, fornece essencialmente especificações e diagramas que perdem utilidade ao se codificar, pois, ao se desenvolver um software, as modificações decorrentes das mudanças no projeto temporalmente não são atualizadas nos modelos iniciais.

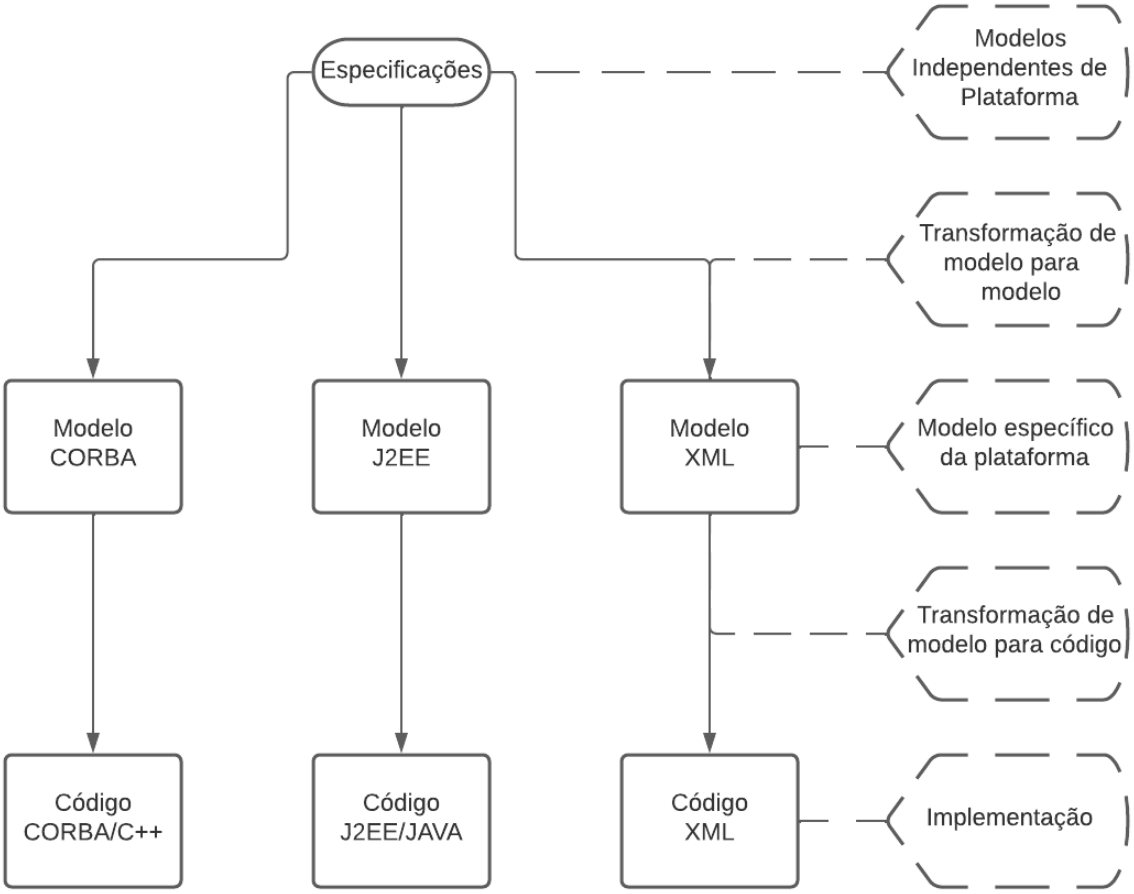
Com intuito de reduzir o tempo e recursos empregados no desenvolvimento de projetos, bem como aumentar a produtividade e qualidade do projeto, se faz necessária a utilização da abordagem dirigida a modelos (MDD - Model Driven Development). A MDD evita discrepância na modelagem, portanto, reduz o tempo de desenvolvimento priorizando a geração de código através da modelagem. Rumpe (2017) considera que, quando se faz a utilização da abordagem de modelagem, torna-se compreensível a visualização do projeto devido aos elementos gráficos que tornam o projeto mais compacto.

Um caso distinto de MDD é a arquitetura orientada por modelos (MDA - Model Driven Architecture). Para Kleppe et al. (2005), MDA é uma estrutura para desenvolvimento de softwares criada pela Object Management Group (OMG - Grupo de Gerenciamento de Objetos), visa estabelecer regras para a estruturação das especificações de um determinado software, ou seja, a MDA é uma visão da MDD desenvolvida pela OMG.

Segundo Völter et al. (2013) as transformações de modelos são uma forma de mapear os respectivos modelos para um próximo modelo ou forma de texto. Usualmente é abordado a transformação de um modelo UML para uma forma de texto por princípios de menor tempo de desenvolvimento do projeto, mas existem casos onde se deseja elaborar uma transformação de metamodelos como, por exemplo, o trabalho de (SOARES; VRANCKEN, 2008) que cria uma abordagem de metamodelagem para

transformar diagramas de sequência UML 2.0 em redes de Petri. A Figura 1 mostra o princípio básico de uma transformação.

Figura 1 – Principio básico da transformação de modelos baseados em UML



Fonte: Modificado de Rosen (2003).

Três abordagens de modelos são exemplificadas na Figura 1. A abordagem de arquitetura de solicitação de objetos comuns (CORBA - Common Object Request Broker Architecture) estruturada pela OMG, foi desenvolvida com intuito de lidar com a construção de um projeto em máquinas com sistemas operacionais diferentes, proporcionando a integração dos projetos baseados nas estruturas da OMG. A plataforma Java 2, edição empresarial (J2EE - Java 2 Platform, Enterprise Edition) que estabelece o desenvolvimento de aplicações corporativas multicamadas, além da linguagem de marcação extensível (XML - Extensible Markup Language) sendo uma linguagem de marcação desenvolvida pelo consórcio da world wide web (W3C - World Wide Web Consortium) que se baseia em um formato de dados organizados de forma hierárquica.

2.3 LINGUAGEM DE MODELAGEM UNIFICADA (UML)

A metodologia para desenvolver um software tem como princípio transcrever, de diversas maneiras, as ideias e o que se deseja obter no fim do projeto, podendo ser feito por meio descritivo ou por imagens. Algumas empresas utilizam a linguagem de notação *Unified Modeling Language* (UML - linguagem de modelagem unificada) que, sendo uma iniciativa desenvolvida pela OMG, é empregada na MDA.

A OMG (2017), segundo sua própria documentação, é uma arquitetura conceitual para um conjunto de especificações de tecnologia, que suporta uma abordagem a modelos para o desenvolvimento de software. O modelo, na qual abstrai todos os detalhes do sistema que podem ser descritos através da linguagem UML. Guedes (2011) afirma que UML é uma linguagem visual com intuito de modelar softwares baseados em um modelo de análise no paradigma da orientação a objetos. Para Silva e Videira (2001), UML é uma linguagem para visualização, construção, especificação e documentação de um sistema de software.

Os diagramas UML podem ser decompostos em dois grandes grupos, os diagramas comportamentais e os diagramas estruturais. Os diagramas estruturais UML definem a estrutura do sistema a ser modelado de forma estática, portanto, a semântica estrutural UML fornece a base para a semântica comportamental, sendo caracterizada na concepção da mudança dos estados do sistema, que é especificado por meio da modelagem (OMG, 2017). De acordo com OMG (2017), os diagramas comportamentais, para os quais compete modelar como os indivíduos de um determinado domínio transitam durante um tempo, não aludem à quantidade de tempo entre eventos, a menos que isso seja especificado na modelagem através de restrições de tempo.

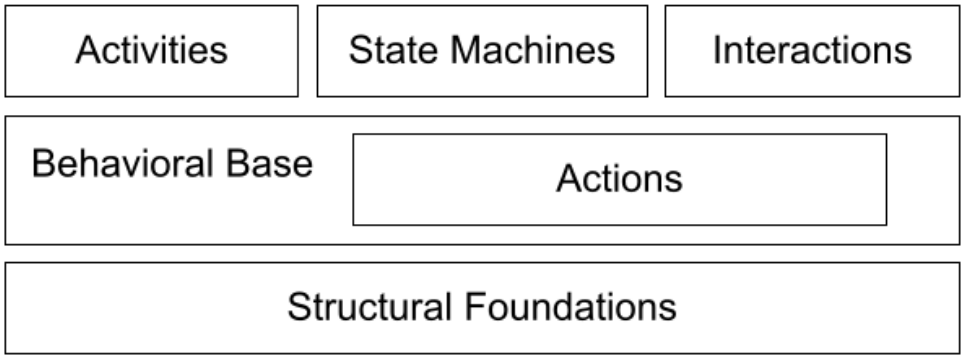
Dentre os diagramas comportamentais existem os diagramas de sequência que representa a ordem de ocorrência entre interações de objetos, diagrama de comunicação que se assemelham aos digramas de sequência, diagramas de máquina de estados que representam de forma gráfica a sequência de um objeto, bem como o evento que causam a transição e o que resulta a transição do estado. Existem também o diagrama de tempo que demonstra o comportamento de um objeto em um determinado período e o diagrama de caso de uso que representam um determinado aspectos do relacionamento do usuário com o sistema. Além deste há o diagrama de atividade (OMG, 2017). No próximo tópico serão demonstrados mais aspectos sobre os diagramas de atividade.

2.3.1 Diagrama de atividade

Os diagramas de atividade estão ligados aos aspectos da base comportamental que está ligada aos fundamentos dos diagramas estruturais como é possível observar na Figura 2, na qual a visualização de classes, ações e atividades são dispostas em

três camadas onde cada camada depende da camada inferior, mas não das superiores. Ou seja, os diagramas de atividade especificam o fluxo de controle e o fluxo de dados entre várias ações que são necessárias para implementar uma atividade.

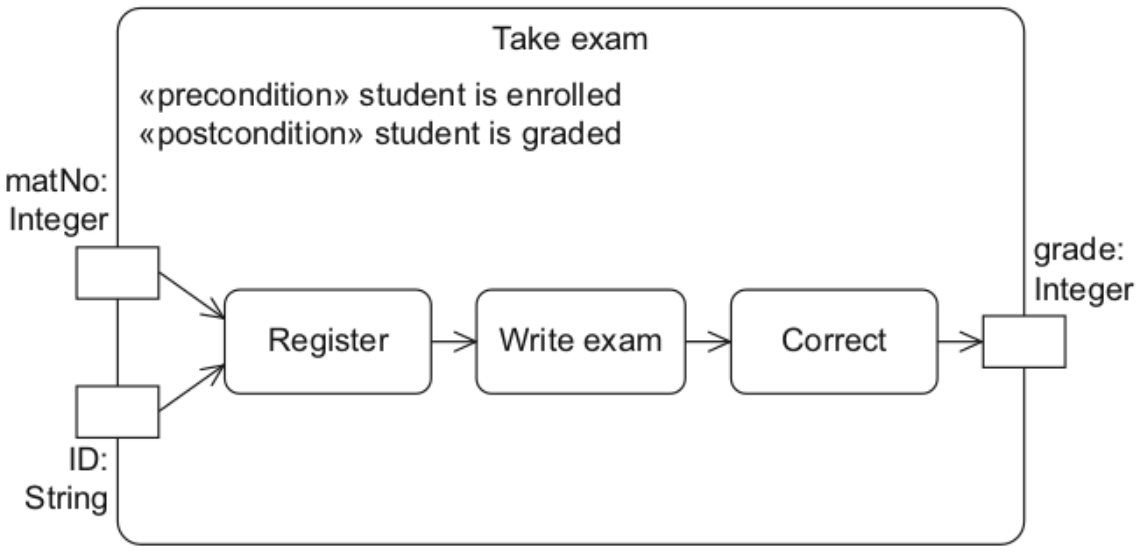
Figura 2 – Esquema das áreas semânticas UML e suas dependências



Fonte: Gessenharter e Rauscher (2011).

Segundo Seidl et al. (2015) um diagrama de atividade permite que seja especificado um comportamento em forma de atividade, onde a representação de uma atividade é dada por um retângulo com cantos arredondados e pode, assim como uma operação, ter parâmetros. Os parâmetros são mostrados como retângulos dispostos no limite da atividade na Figura 3. Sendo possível também especificar as pré-condições e as pós-condições para uma atividade, estas sendo usadas para identificar quais condições devem ser cumpridas antes e após a atividade ser executada (SEIDL et al., 2015).

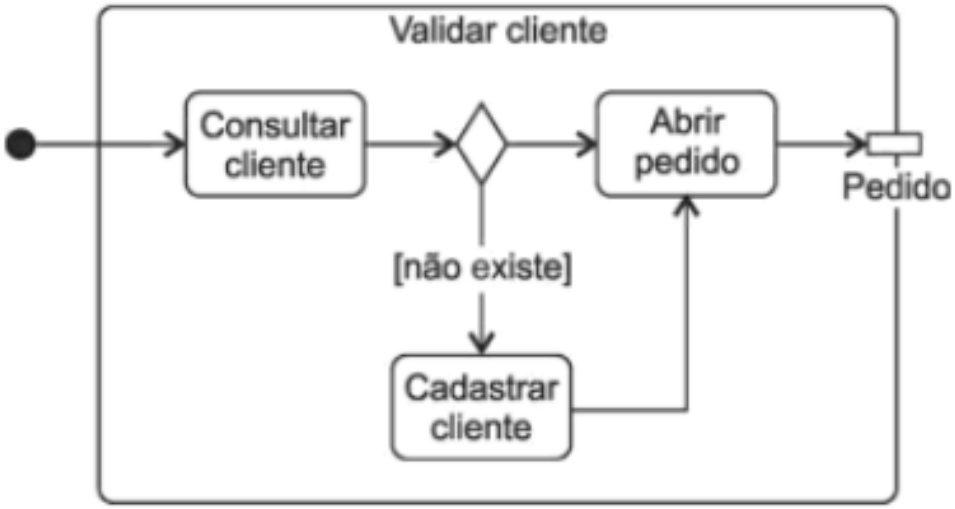
Figura 3 – Exemplo de um diagrama de atividade



Fonte: Seidl et al. (2015).

Outro exemplo é mostrado na Figura 4, neste exemplo se idealiza a criação de um sistema para venda de um produto. O primeiro ponto a se verificar neste diagrama é a criação da atividade validar cliente que contém um conjunto de nós, caracterizados usualmente por ações, onde o fluxo dos nós especifica um comportamento. A atividade deve definir as ações que o sistema deve comportar, sendo elas, consultar o cliente, abrir o pedido e cadastrar o cliente caso ele não tenha cadastro.

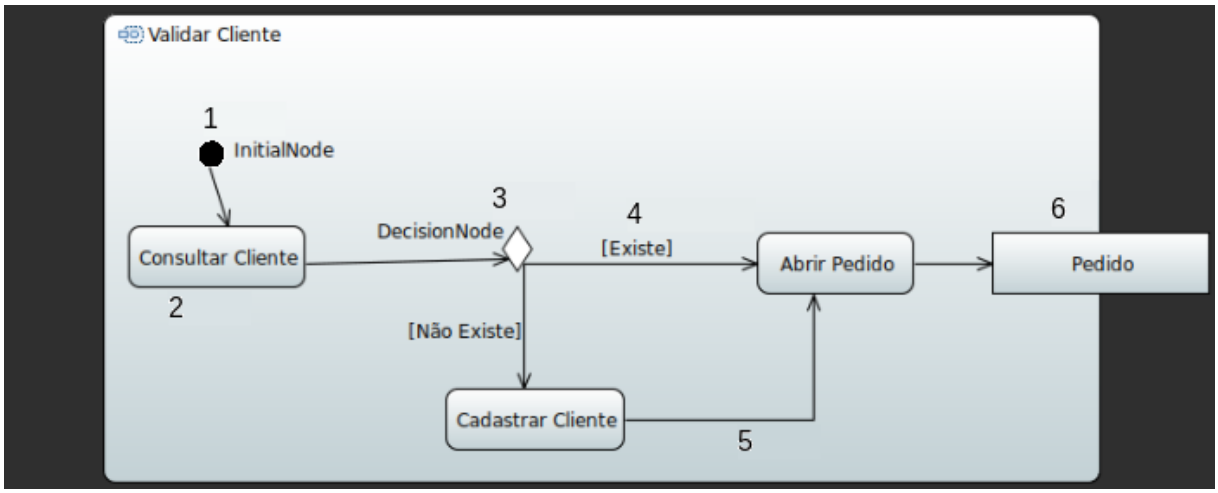
Figura 4 – Exemplo de diagrama de atividade - validar cliente



Fonte: Lima (2011).

Uma aplicação desenvolvida por meio do Eclipse Modeling pode ser observado na Figura 5, todo conceito detalhado na Figura 5 é disponibilizado pela OMG (2017) na documentação da UML.

Figura 5 – Modelagem do diagrama de atividade no Eclipse Modeling



Fonte: Autor.

O primeiro nó a ser demostrado está enumerado como 1, o nó inicial

(*initialNode*) denota o início do fluxo de controle, ou seja, indica o início do diagrama de atividade. O nó representado pelo número 5 denota o controle de fluxo (*Control Flow*), e uma aresta que inicia um nó de atividade após o término do anterior. Posteriormente no número 2 podemos notar o nó de ação opaca (*opaqueAction*) que tem semântica específica de implementação, ou seja, contêm uma estrutura de código de programação preestabelecida na ação.

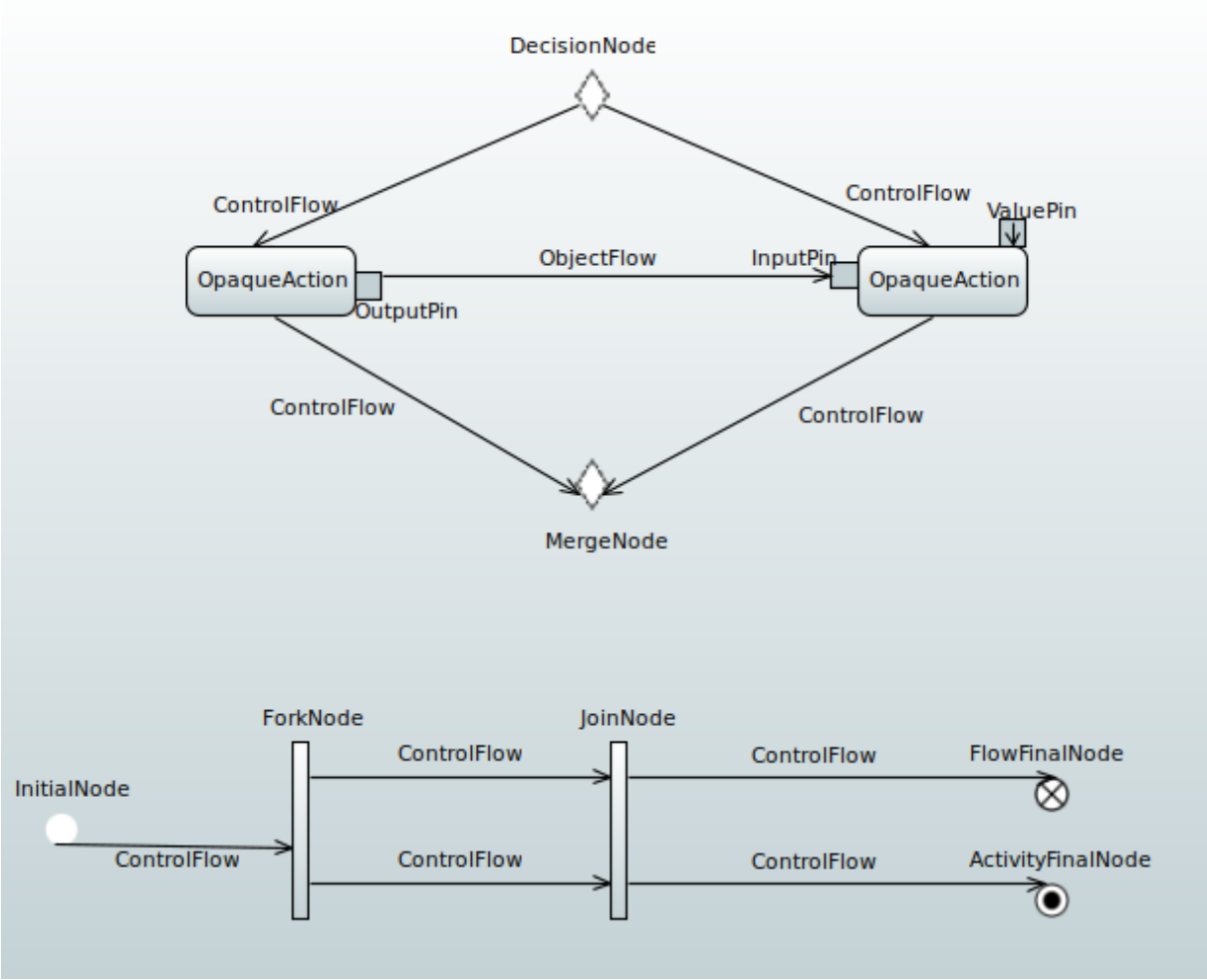
O próximo nó denotado pelo número 3, denominado nó de decisão (*DecisionNode*), é um nó de controle que escolhe entre fluxos de saída, ou seja, dada condição de guarda determina o sentido do controle de fluxo do diagrama. A guarda é ressaltada pelo número 4 e visa permitir a transição do fluxo de controle somente se a condição assumida pela expressão é verdadeira, geralmente atribui-se à guarda valores referentes à expressão booleana. Por fim o nó representado pelo número 6 um nó de parâmetro de atividade(ActivityParameterNode), sendo um nó que contém um objeto para entradas e saídas de atividades.

Outros nós de controle podem ser observados na Figura 6, sendo que os nós de controle são caracterizados por indicar o fluxo de controle de uma atividade, por meio da aresta de controle de fluxo *ControlFlow* Abstraindo assim o raciocínio daquele que modela o diagrama de atividade, e permitindo que as atividades presentes na modelagem tenha comunicação por meio dos fluxos de controle e objeto. Este trabalho não aborda os nós *ForkNode* e *JoinNode* por se tratarem de nós de controle para tarefas em paralelo e também não abordando o *FlowFinalNode*. Na Figura 6 é possível verificar também um pino de saída denominado *OutputPin*, é um pino que contém valores de saída produzidos por uma determinada ação (OMG, 2017).

Há ainda dois outros pinos que podem ser observado na Figura 6, sendo eles o pino de entrada e o pino de valor. O pino de entrada *InputPin* é caracterizado por receber um valor de entrada de uma ação (OMG, 2017). O pino de valor *ValuePin* é um pino de entrada que fornece um valor especificado (OMG, 2017). Todo pino, seja de entrada ou saída de um valor, tem seu sentido de transição estipulado pelo fluxo de objeto. O fluxo de objeto *ObjectFLow* indica o sentido de movimento por onde o valor segue durante a atividade (OMG, 2017).

Além disso, é possível verificar os nós de controle como o nó de decisão *DecisionNode* que por uma guarda faz a ramificação dos fluxos conforme a decisão, a guarda que predispõe uma condição para a ramificação (OMG, 2017). Portanto, o nó de decisão é um nó de controle que dispõe o sentido do fluxo de controle conforme uma condição predisposta. Após as ramificações do nó de decisão há sempre para esta abordagem de modelo um nó de junção *MergeNode*, esse nó reune vários fluxos alternativos. A Figura 6 ainda traz dois nós de controle essenciais o nó inicial *InitialNode* e o nó de atividade final *ActivityFinalNode*, onde cada um respectivamente indica o início do diagrama de atividade e o final do mesmo (OMG, 2017).

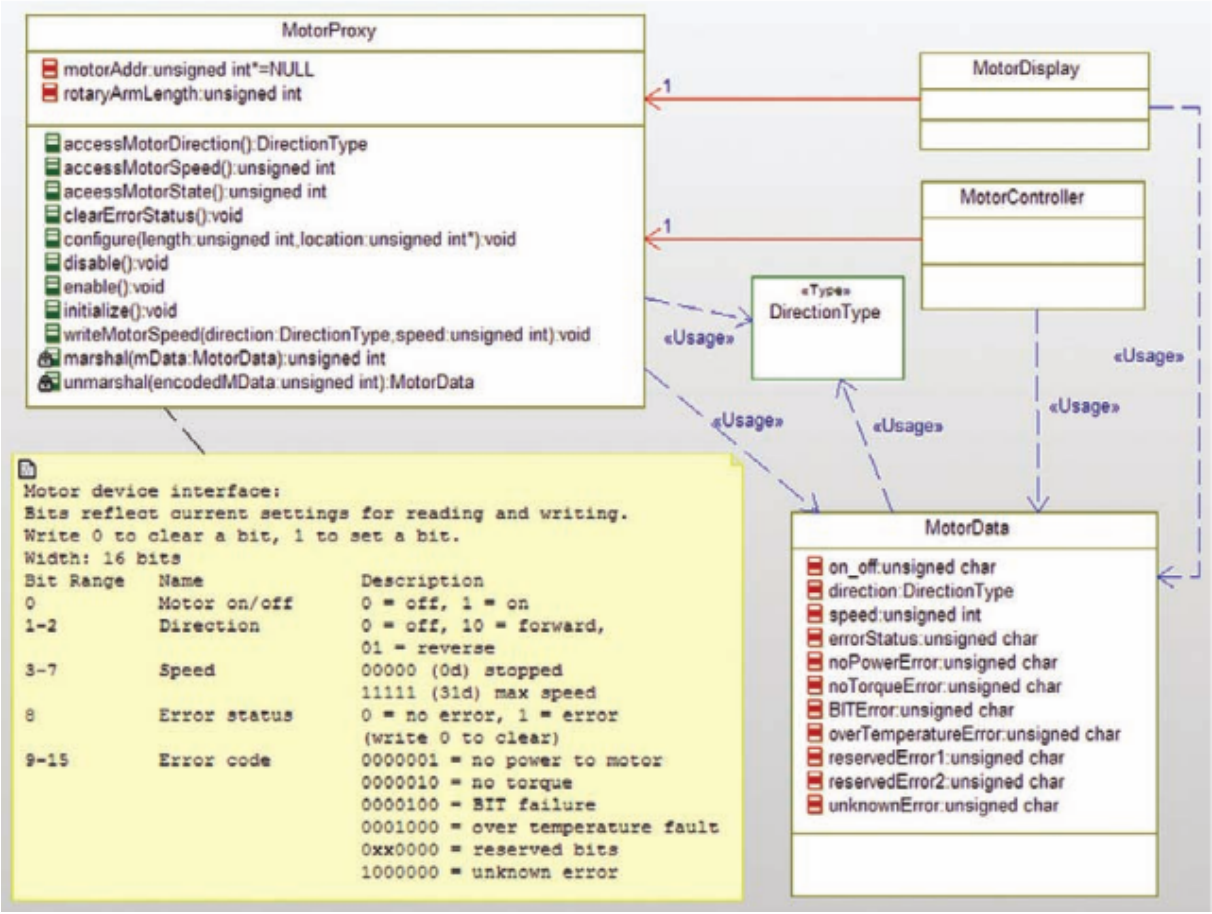
Figura 6 – Nós de controle



Fonte: Autor.

Um exemplo de diagrama de classe pode ser visto na Figura 7 representando um sistema de motor, com uma interface de 16 bits, onde há a classe que abstrai o proxy do motor e de dois clientes. Para cada método da classe *MotorProxy* pode ser associado um diagrama de atividade, com os nós e ações de atividade disponíveis na Figura 8, sendo que cada nó e ação contida na Figura 8 contém uma característica única que contribui para a modelagem, seguindo a abstrações da documentação disponibilizada pela OMG (OMG, 2017) que detalha a UML 2.5.1.

Figura 7 – Exemplo diagrama de classe de proxy de hardware

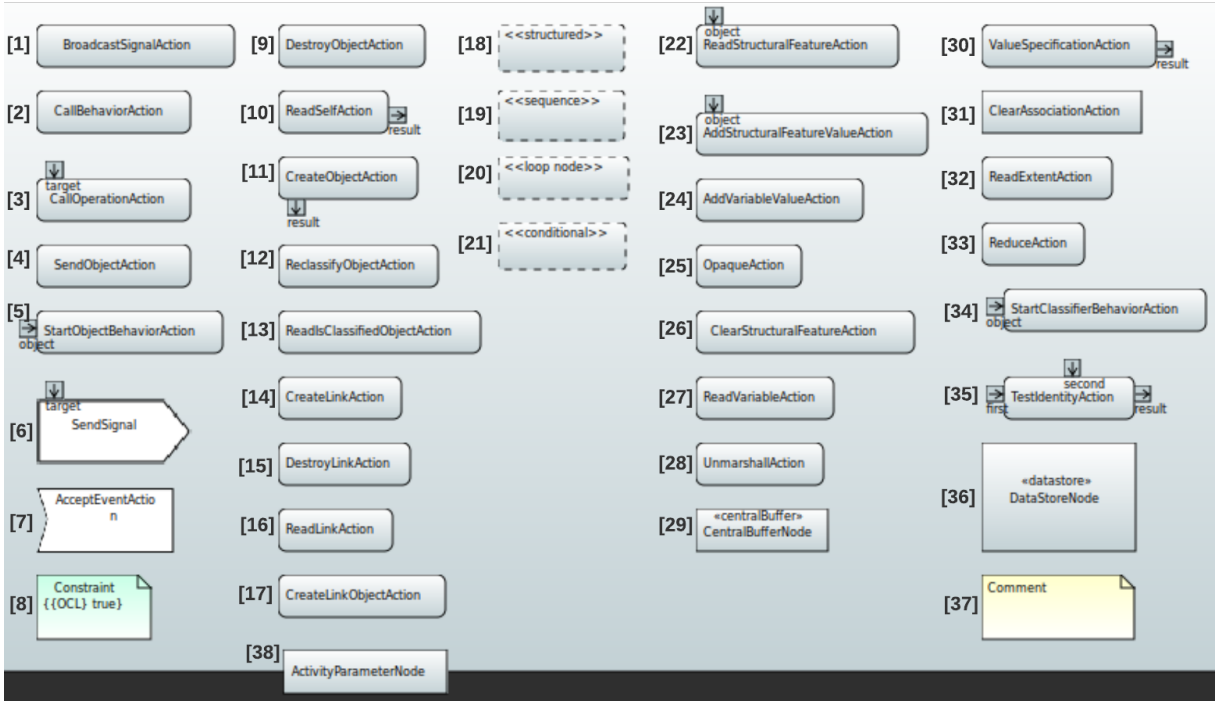


Fonte: Douglass (2011).

Diversas ações são disponibilizadas pela OMG (OMG, 2017), como é possível verificar na Figura 8, onde o número 1 representa uma ação que gera um sinal de transmissão *BroadcastSignalAction*, que transmite um sinal para todos os possíveis alvos do sistema. Essa ação tem em suas propriedades a característica de estipular um sinal e uma porta opcional do objeto receptor em que um recurso comportamental é invocado (OMG, 2017). A ação 2 é uma chamada de comportamento *CallBehaviorAction*, sendo os valores de argumento transmitidos nos parâmetros de entrada do comportamento invocado. A ação *CallBehaviorAction* pode operar no modo síncrono e assíncrono, e em suas propriedades consta um campo para especificar o comportamento e uma porta opcional para invocar um recurso comportamental (OMG, 2017).

A ação 3 é uma chamada de operação *CallOperationAction*, que conta com uma porta opcional que invoca um recurso comportamental, também contendo um pino de entrada e um campo para selecionar a operação que será invocada (OMG, 2017). A ação 4 envia um objeto de entrada para um objeto de destino *SendObjectAction*, esta ação conta com um pino de entrada e um campo para o objeto a ser transmitido para o objeto de destino, além de uma porta opcional que invoca um recuso comportamental

Figura 8 – Nós de atividade e ações



Fonte: Autor.

(OMG, 2017). A ação 5 corresponde a uma ação que inicia um comportamento de objeto *StartObjectBehaviorAction*, ou seja, inicia a execução de um comportamento instanciado diretamente ou do comportamento do classificador de um objeto, assim como a opção de uma porta opcional (OMG, 2017).

A ação 6 é caracterizada por enviar um sinal de ação *SendSignalAction*, e criando uma instância do sinal transmitido para o objeto de destino, onde o pino de entrada é utilizado para fornecer valores para os atributos do sinal, também conta com a opção de uma porta opcional (OMG, 2017). Já a ação 7 *AcceptEventAction* aguarda a ação de um ou mais eventos específicos. A ação 8 é caracterizada por restringir expressamente por meio de texto em linguagem natural ou em linguagem legível por máquina um elemento (OMG, 2017).

A ação numero 9 destrói um objeto *DestroyObjectAction*, sendo necessário indicar o objeto a ser excluído em sua propriedade (OMG, 2017). A ação de auto leitura 10 *ReadSelfAction* realiza a leitura do objeto onde o contexto do comportamento está sendo executado, e indica através de um pino de saída (OMG, 2017). O número 11 condiz com a ação de criar um objeto *CreateObjectAction*, tendo em suas propriedades um pino de saída resultante em um classificador (OMG, 2017). A ação de reclassificação de objeto *ReclassifyObjectAction*, número 12, visa alterar o classificador de um objeto (OMG, 2017).

A ação 13 realiza a leitura de um objeto classificado *ReadIsClassifiedObjectAction*, e determina se esse objeto é classificado por

um determinado classificador (OMG, 2017). A ação *ReadIsClassifiedObjectAction* contém em suas propriedades a possibilidade de indicar um objeto, o classificador a ser analisado e um pino de saída com o resultado da análise. Já a ação 14 cria uma conexão *CreateLinkAction* entre os links criados (OMG, 2017). A ação de número 15 destrói a conexão *DestroyLinkAction* entre os links criados(OMG, 2017). A ação 16 realiza a leitura de uma conexão *ReadLinkAction*, e por um pino de saída obtêm-se o resultado (OMG, 2017).

A ação 17 cria objetos de conexão *CreateLinkObjectAction*, tendo em suas propriedades a possibilidade de criar um pino de saída para obter esse objeto (OMG, 2017). O número 18 é um nó de atividade estruturada *StructuredActivityNode* (OMG, 2017). O nó de sequência *SequenceNode*, número 19, é um nó de atividade estruturada que executa uma sequência de nós em ordem (OMG, 2017). Já o número 20 é um nó de laço *LoopNode* , sendo tambem um nó de atividade estruturada que representa um laço iterativo com um pino de saída no fragmento de teste, cujo valor é examinado para determinar se o laço deve ser executado (OMG, 2017). O nó condicional *ConditionalNode*, número 21, também é um nó de atividade estruturada que escolhe uma entre algumas coleções alternativas de nós executáveis para executar. (OMG, 2017).

A ação 22 realiza a leitura de um recurso estrutural *ReadStructuralFeatureAction*, ou seja, é uma ação de recurso estrutural que recupera os valores de um recurso estrutural (OMG, 2017). O *ReadStructuralFeatureAction* tem como característica em suas propriedades a possibilidade de um pino de saída com a leitura e um pino de entrada que recebe o objeto estrutural. A ação 23 adiciona um valor estrutural *AddStructuralFeatureValueAction* para adicionar um recurso estrutural (OMG, 2017). A ação 24 adiciona um valor a uma variável *AddVariableValueAction* (OMG, 2017).

A ação opaca *OpaqueAction*, número 25, é utilizada para inserir um trecho de código pré-estabelecido na modelagem, sendo possível estipular a linguagem de programação e o trecho do código em suas propriedades (OMG, 2017). A ação 26 limpa um recurso estrutural *ClearStructuralFeatureAction*, ou seja, remove todos os valores de um recurso estrutural.(OMG, 2017). A ação 27 le um variável *ReadVariableAction* e retorna através de um pino de saída (OMG, 2017). A ação 28 recupera os valores *UnmarshallAction* das características estruturais de um objeto e os coloca em pinos de saída (OMG, 2017).

O nó 29 é um nó de buffer central *CentralBufferNode* que gerencia fluxos de várias origens e destinos.(OMG, 2017). A ação 30 especifica um valor *ValueSpecificationAction* por um valor pré-estabelecido em suas propriedades (OMG, 2017). Já o número 31 é uma ação que realiza a limpeza da associação *ClearAssociationAction*, ou seja, é uma ação que destrói todos os *links* de associação

de um determinado objeto. (OMG, 2017). A ação 32 efetua a leitura de uma extensão *ReadExtentAction*, portanto, lê as instâncias de um classificador.(OMG, 2017). Já a ação 33 realiza a redução *ReduceAction*, reduzindo assim um conjunto de um único valor repetido usando redutor comportamental (OMG, 2017).

O número 34 é uma ação que inicia um comportamento do classificador *StartClassifierBehaviorAction* do objeto que é recebido pelo pino de entrada (OMG, 2017). A ação 35 realiza o teste de um identificador *TestIdentityAction* assim verificando se dois objetos dos pinos de entrada são idênticos, obtendo um resultado no pino de saída (OMG, 2017). O nó 36 é denominado nó de armazenamento de dados *DataStoreNode*, sendo uma forma de *CentralBufferNode* para dados constantes (OMG, 2017). Já o comentário identificado pelo número 37 *Comment* é uma anotação textual que pode ser acrescentada a um conjunto de elementos. (OMG, 2017). E, por fim, a Figura 8 traz o nó de parâmetro de atividade *ActivityParameterNode*, número 38, que fornece valores de entrada e saída do diagrama de atividade, o qual suas propriedades é possível configurar entre entrada, saída ou retorno. Além de poder especificar o tipo do objeto e os tonks para a borda de saída (OMG, 2017).

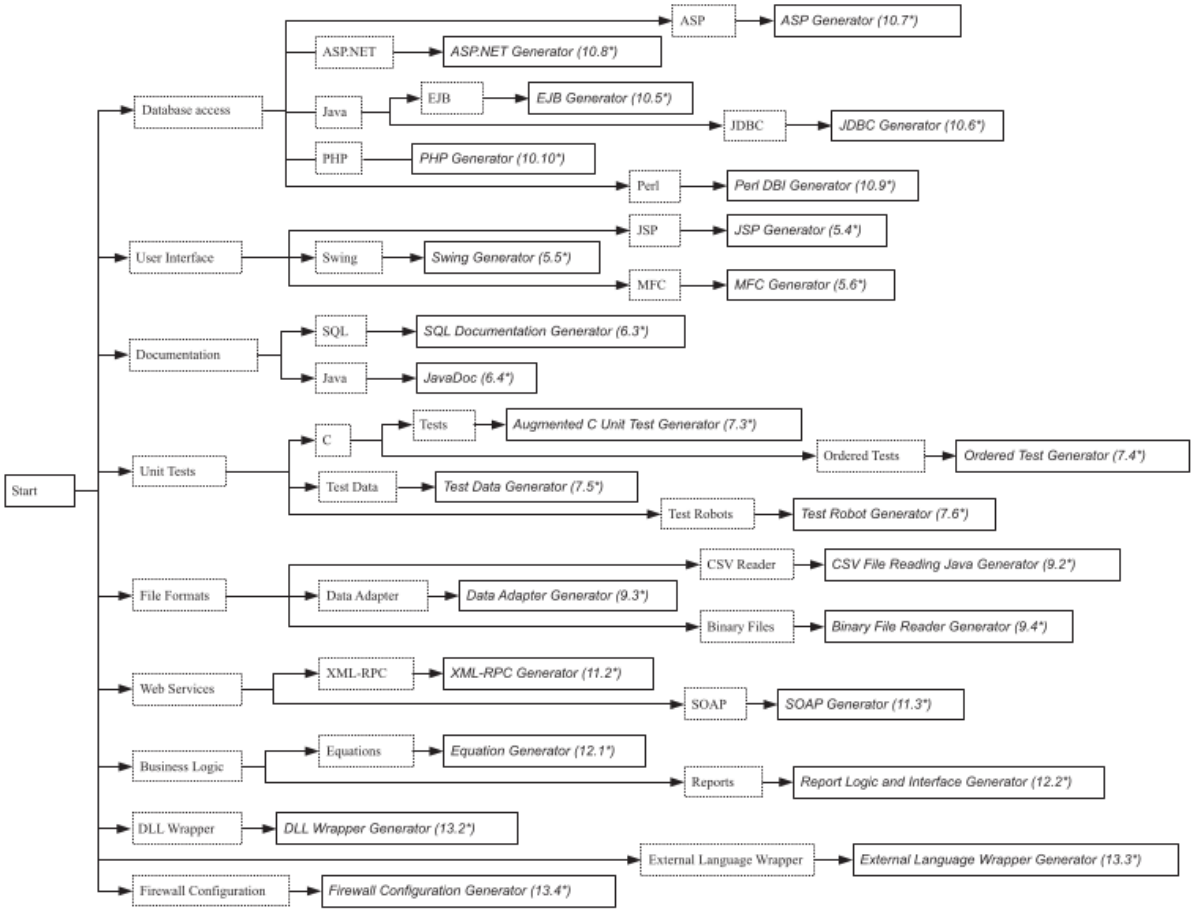
2.4 GERAÇÃO DE CÓDIGO

O método para geração código é uma técnica semelhante aos padrões de design orientado a objeto, pela 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). Quando se busca-se implementar vários modelos de código com estruturas comuns por um longo período, o desenvolvimento de um código que consiga gerar outros códigos é necessário. A visão de Herrington (2003) sobre a estrutura de geração de código através de uma árvore pode ser observada na Figura 9.

Rumpe (2017) afirma que o processo de gerar código auxilia no desenvolvimento e gerenciamento de possíveis complexibilidades que possam aparecer durante a criação de um software, se fazendo ainda mais presente nos projetos atuais devido à modernização das ferramentas de desenvolvimento de softwares. Conforme Herrington (2003), durante o desenvolvimento de grandes quantidades de códigos produzidos a mão, há uma perda da qualidade devido a abordagens que possam mudar no decorrer do projeto, porém, quando as alterações são feitas no gerador, seja para corrigir ou melhorar algo, toda a base do código é prontamente alterada, ou seja, o indicado é utilizar um computador para facilitar o trabalho do desenvolvedor.

Para Rumpe (2017), a geração de um código executável a partir de um modelo

Figura 9 – Árvore de geração de código



Fonte: Herrington (2003).

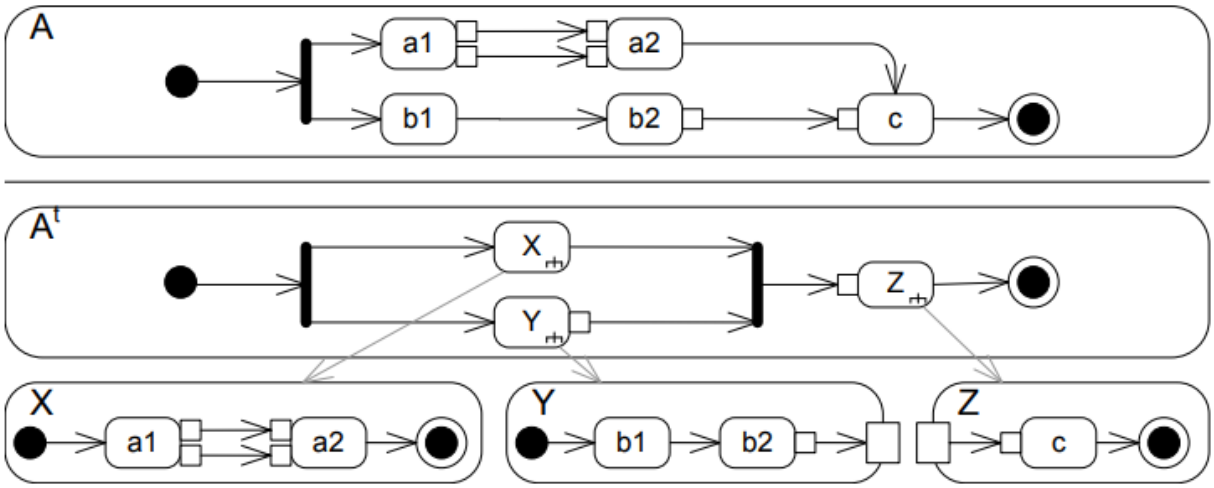
UML que, essencialmente, representa uma redução ou abstração de um sistema em relação à escala, nível de detalhamento ou funcionalidade, é um grande desafio. Isso, não se aplica unicamente para as ferramentas baseadas em UML, mas também para ferramentas de modelagem similares, como Simple DirectMedia Layer (SDL), Autofocus e Statemate. Contudo, em contraponto ao desafio, há um ganho plausível de eficiência com atenuação na quantidade de código a ser escrito, verificado e testado manualmente.

Segundo Herrington (2003), uma das principais vantagens de desenvolver um gerador de código é poder controlar aspectos do desenvolvimento da ferramenta, bem como evitar erros gerados pela codificação manual. Deste modo, Herrington (2003) define como uma desvantagem do gerador de código o treinamento necessário para operar a ferramenta, tal como a necessidade de atualizar a ferramenta futuramente. Por mais que a ferramenta de geração de código traga um retorno rápido e prático, há um contraponto na inflexibilidade referente a estrutura e aspectos relacionados a modelagem.

2.5 TRABALHOS RELACIONADOS

O artigo Gessenharter e Rauscher (2011) apresenta uma abordagem para geração de código usando diagramas de atividades precedidas por transformações de modelo. A abordagem proposta no artigo se concentra em três conceitos *ObjectFlow* (fluxos de objetos), simultaneidade e *InterruptibleActivityRegions* (Região de atividade interruptível). Ou seja, tornar evidente as bifurcações e junções explicitas, mover as sequências para separar comportamentos e substituir por *CallBehaviorActions* e se possível substituir os nós de controle por um único, sendo essas as principais etapas a serem aplicadas para obter uma melhora na geração de código como ilustrado na Figura 10. Os autores informam que a abordagem utilizada tem limitação com diagramas de atividade que podem ser complexos na utilização de nós de controle.

Figura 10 – Atividade A e sua transformação At correlacionadas com atividades X, Y e Z

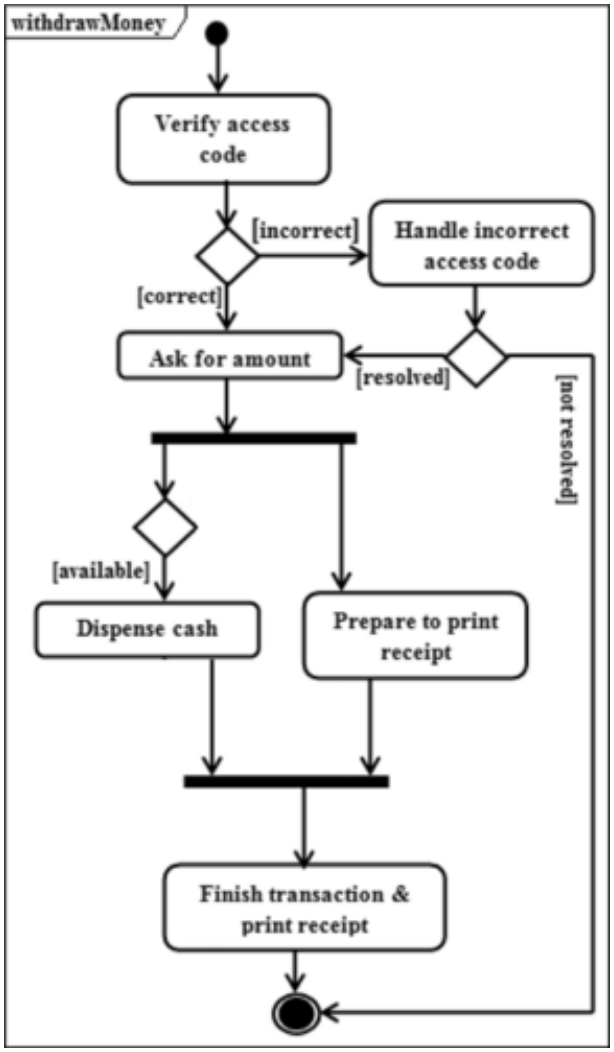


Fonte: Gessenharter e Rauscher (2011).

O artigo de Sunitha e Samuel (2018) examina como melhorar a geração de código a partir de modelos UML, com a ajuda da Object Constraint Language explorando as possibilidades de incorporar OCL em modelos de atividades UML, portanto, gerando código a partir dos diagramas de atividades aprimorados com OCL. Expressões OCL são adicionadas como parte dos modelos dos diagrams de atividade UML para melhorar a geração de código e melhor especificar o comportamento. A ferramenta implementada com base no método proposto apresenta um resultado promissor. Mais de 80% do código-fonte é gerado usando a ferramenta. O metamodelo proposto no artigo fornece uma boa base teórica para anexar instruções OCL a cada elemento nos diagramas de atividades UML. Dado que UML e OCL são comumente usados na indústria de software, o método proposto de geração de código auxilia na melhora e aumenta a produtividade das indústrias de software, pois reduz o esforço e o tempo de desenvolvimento de software. É possível verificar a diferença nas Figuras 11 e 12. Onde a Figura 12 traz

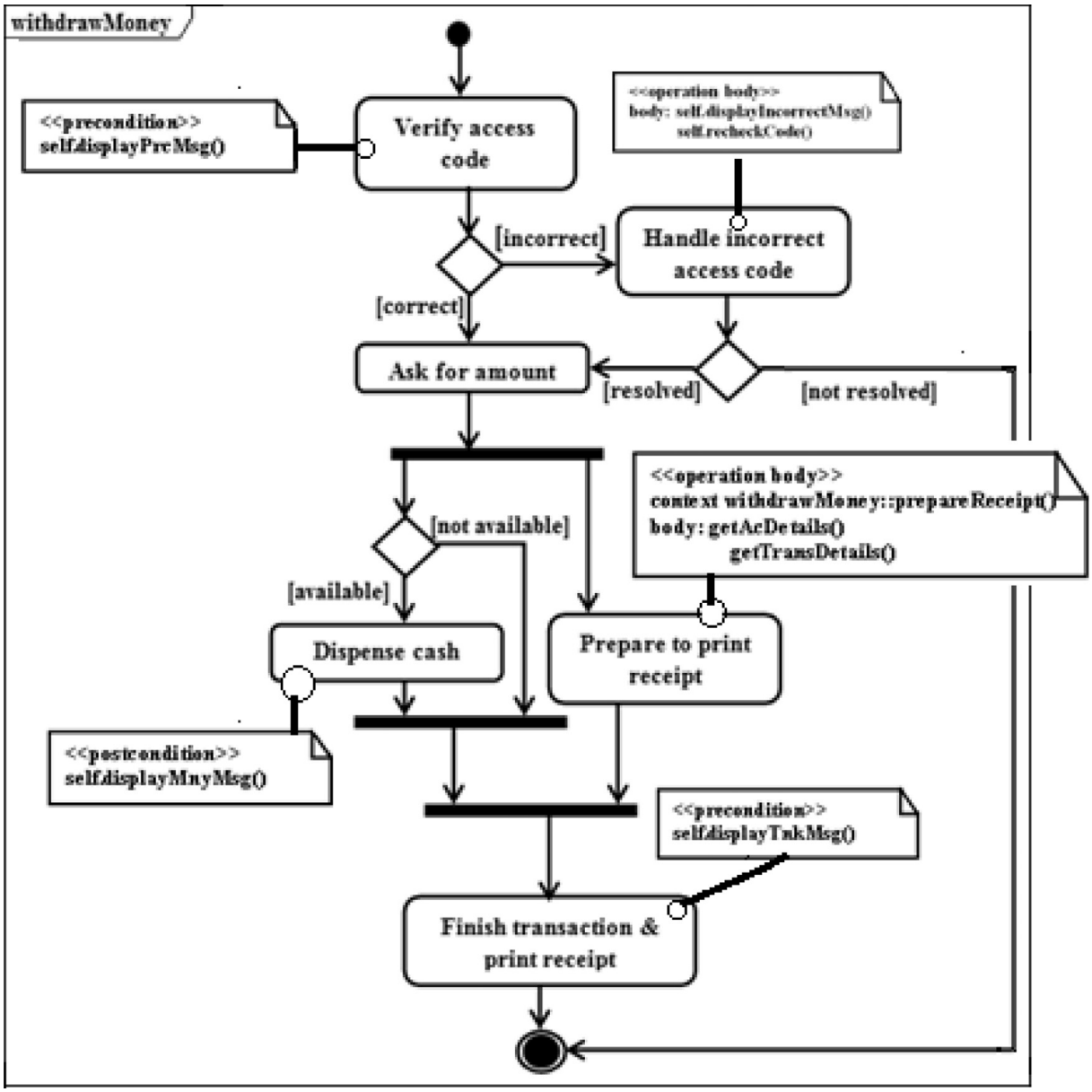
consigo a OCL, portanto as restrições do objeto.

Figura 11 – Diagrama de atividades para máquina ATM sem OCL



Fonte: Sunitha e Samuel (2018).

Figura 12 – Diagrama de atividades para máquina ATM com OCL



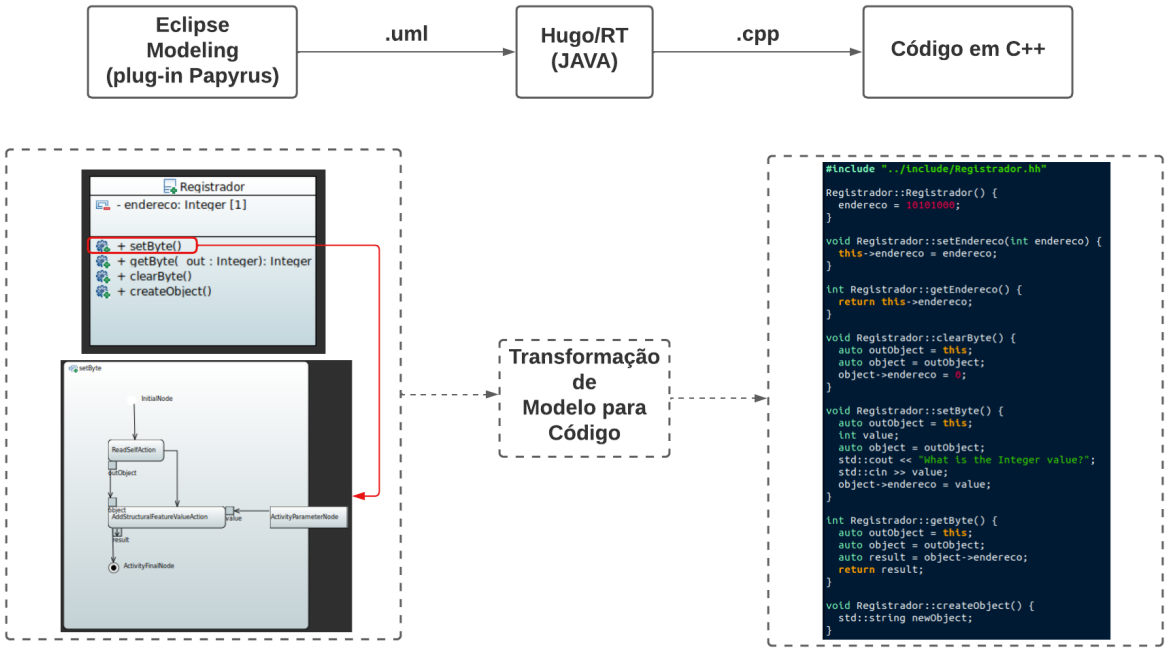
Fonte: Sunitha e Samuel (2018).

3 MATERIAIS E MÉTODO

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 começou 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.

Posteriormente um arquivo *.uml* é gerado por meio desta modelagem, e este é interpretado pela abordagem do programa Hugo-RT (KNAPP, 2022), sendo a ferramenta baseada na linguagem de programação Java, onde o programa Hugo-RT contém a funcionalidade para gerar código de diagrama de classe UML, assim esse trabalho visa acrescentar uma abordagem para geração de código por meio de diagramas de atividades UML, gerando código na linguagem C++ por uma abordagem de diagrama de atividade para código, na sequência apresentada pela 13.

Figura 13 – Abordagem Aplicada para Geração de Código

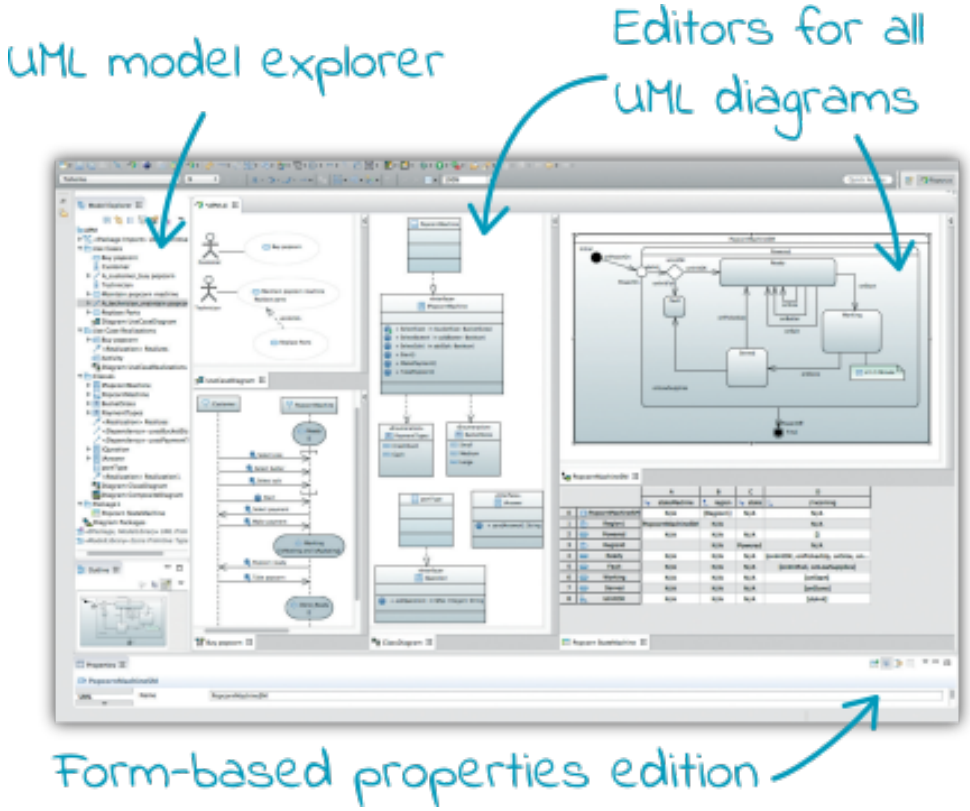


Fonte: Autor.

3.1 ECLIPSE PAPYRUS

O ambiente de modelagem Eclipse Papyrus (PROJECT, 2022) é uma ferramenta independente que pode ser utilizada como plug-in no software Eclipse Modeling (PROJECT, 2022).O Papyrus é uma ferramenta de modelagem que oferece suporte para UML2 conforme definido pela OMG, SysML e para sistemas em tempo real através da UML-RT. A Figura 14 exemplifica as funcionalidades da UML2 disponíveis no Papyrus para modelagem.

Figura 14 – Funcionalidades do Papyrus para UML2



Fonte: Foundation (2022).

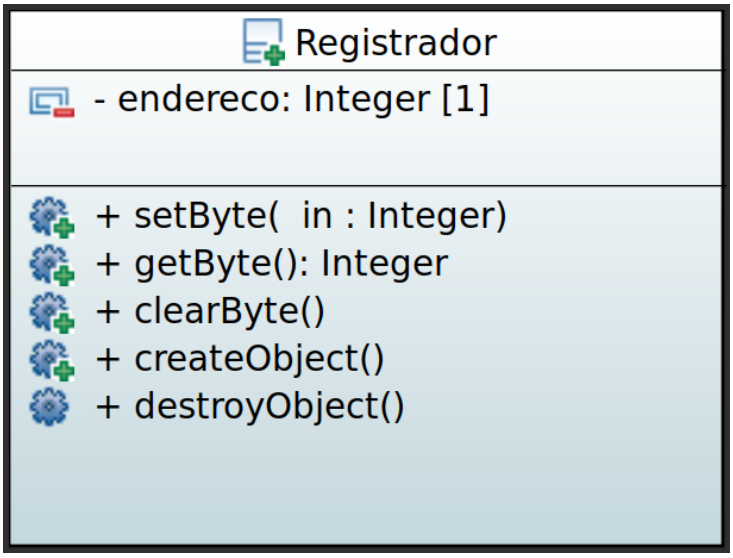
Outros exemplos podem ser observados nas figuras 15, 16, 17, 18 e 19 que serão melhor exemplificadas nos próximos tópicos, sendo estas modelagens geradas por meio da ferramenta Papyrus Modeling. Assim , o Papyrus é uma ferramenta de modelagem que abstrai as características da UML2, auxiliando na configuração das propriedades da modelagem e das representações gráficas dos conjuntos de nós e arestas estipuladas pela OMG.

3.2 MODELAGEM

Inicialmente é necessário estruturar a abordagem de modelagem e seus aspectos de configuração, para deste modo conduzir uma abordagem de modelagem

coerente. Através do Eclipse Modeling (PROJECT, 2022) é criada uma estrutura de modelagem de classe por meio do plug-in Papyrus denominado Registrador como é possível verificar na Figura 15. Esta classe é caracterizada por um atributo denominado endereço que armazena um valor inteiro de 8 bits, e pelos métodos setByte, getByte, clearByte e createObject.

Figura 15 – Classe Registrador



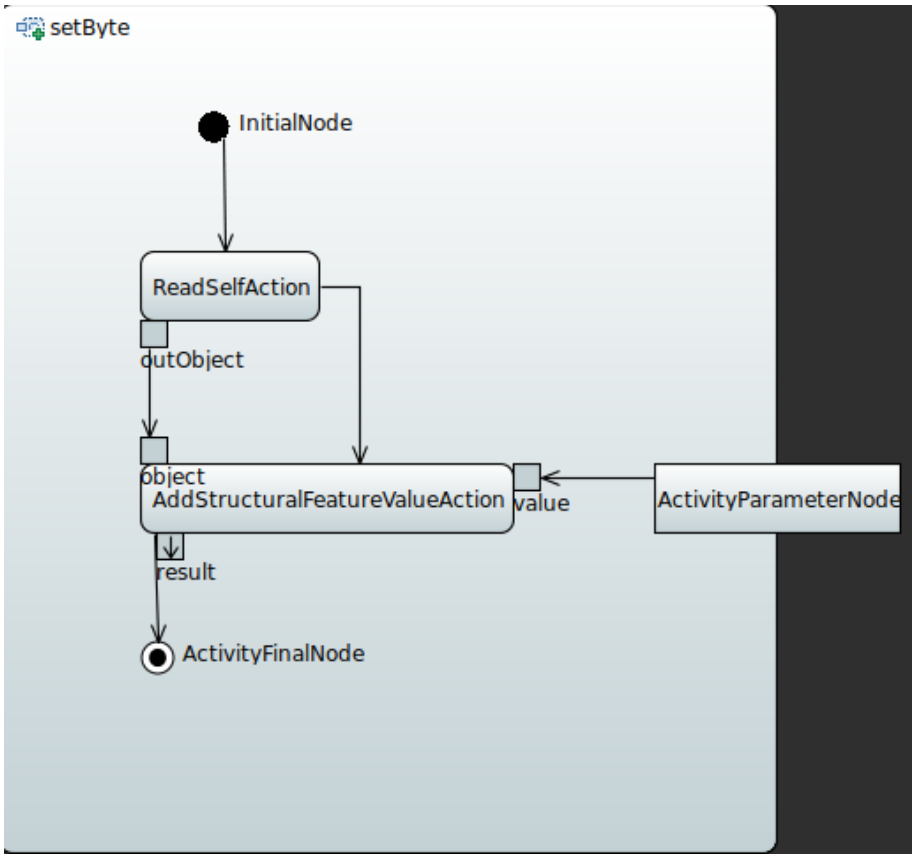
Fonte: Autor.

Para cada método foi atribuído um diagrama de atividade UML, onde a Figura 16 mostra o diagrama de atividade setByte. Esse diagrama consiste em um nó inicial (*InitialNode*), que por um controle de fluxo *(ControlFlow*) transita para um nó que faz a leitura de uma auto-ação (ReadSelfAction) que acessa o objeto da classe em execução. Em seguida há uma transição do controle de fluxo (*ControlFlow*) e de controle de objeto (*ObjectFLow*) para um nó que adiciona um valor ao recurso estrutural (*AddStructuralFeatureValueAction*), recebendo esse valor por meio do nó parâmetro de atividade (*ActivityParameterNode*). Prossegue para uma transição de um fluxo de controle (*ControlFlow*) para o nó que indica o fim da atividade (*ActivityFinalNode*).

Na Figura 17 é possível observar o diagrama de atividade getByte. Esse diagrama consiste em um nó inicial (*InitialNode*), que por um controle de fluxo (*ControlFlow*) transita para um nó que faz a leitura de uma auto-ação (*ReadSelfAction*) que acessa o objeto da classe em execução. Posteriormente há uma transição do controle de fluxo (*ControlFlow*) e de controle de objeto (*ObjectFLow*) para um nó que realiza a leitura de recurso estrutural (*ReadStructuralFeatureAction*). Por fim há duas transições, uma do fluxo de controle (*ControlFlow*) para o nó que indica o fim da atividade (*ActivityFinalNode*), e a outra transição é um controle de objeto que flui até um nó de parâmetro de atividade (*ActivityParameterNode*). O nó de parâmetro de atividade (*ActivityParameterNode*) é caracterizado por aceitar valores dos parâmetros

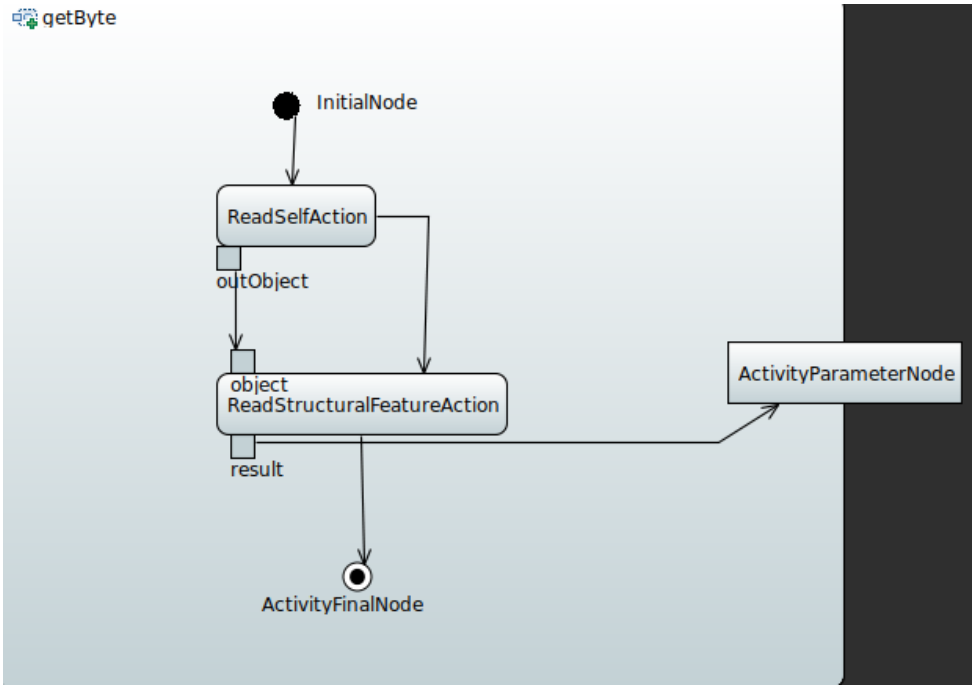
de entrada ou fornecer valores para os parâmetros de saída de uma atividade.

Figura 16 – Diagrama de atividade setByte



Fonte: Autor.

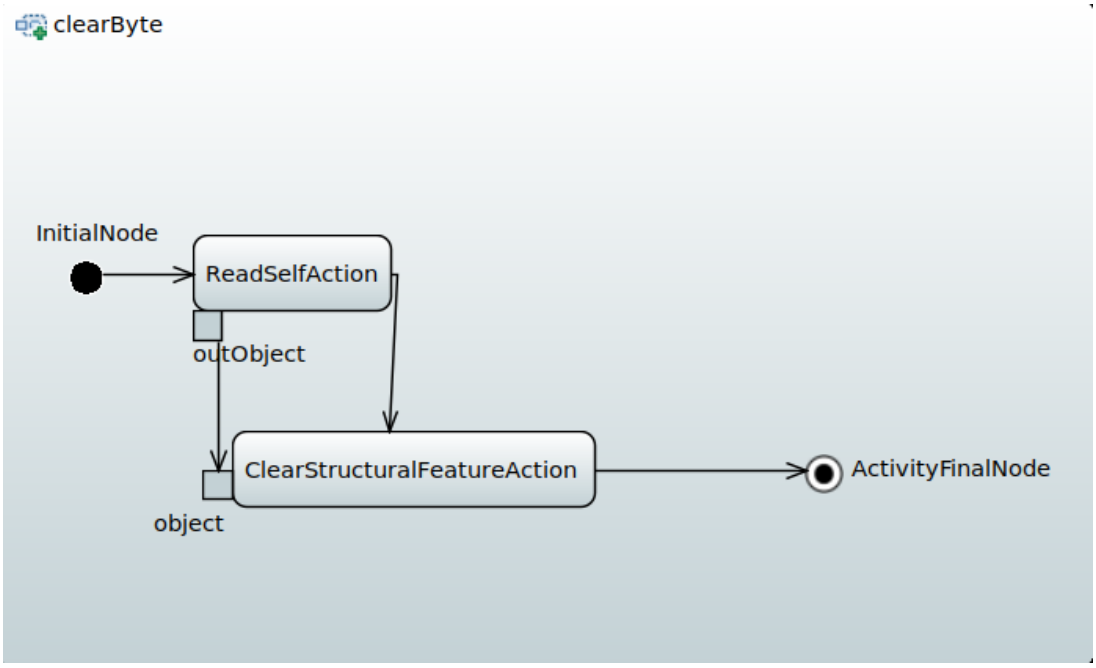
Figura 17 – Diagrama de atividade getByte



Fonte: Autor.

Na Figura 18 é possível observar o diagrama de atividade clearByte. Esse diagrama consiste em um nó inicial (*InitialNode*), que por meio de um controle de fluxo (*ControlFlow*) transita para um nó que faz a leitura de uma auto-ação (*ReadSelfAction*) que acessa o objeto da classe em execução. Posteriormente há uma transição do controle de fluxo (*ControlFlow*) e de controle de objeto (*ObjectFLow*) para um nó que limpa um recurso estrutural (*ClearStructuralFeatureAction*), seguindo com intuito de uma transição de um fluxo de controle (*ControlFlow*) para o nó que indica o fim da atividade (*ActivityFinalNode*).

Figura 18 – Diagrama de atividade clearByte



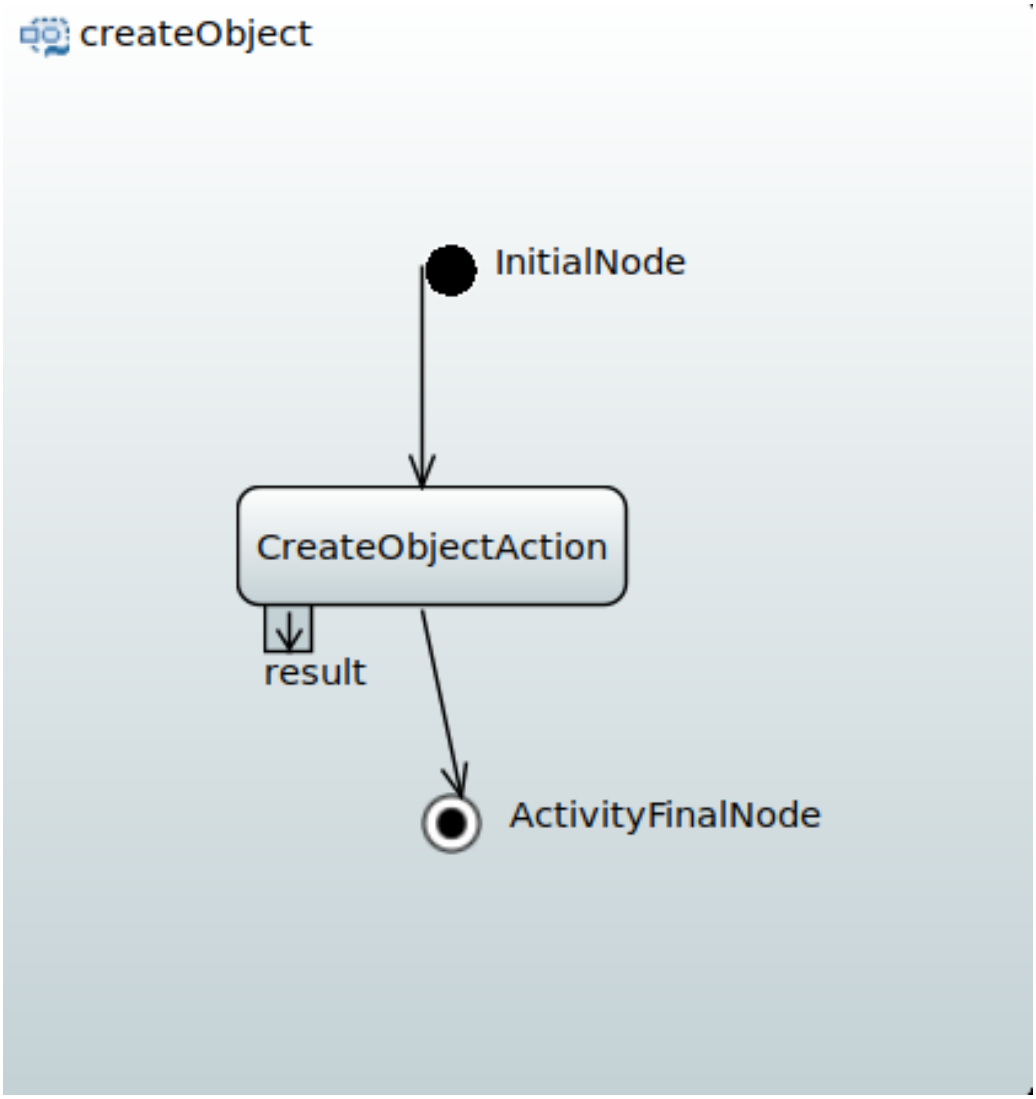
Fonte: Autor.

Na Figura 19 é possível observar o diagrama de atividade createObject. Consistindo em um nó inicial (*InitialNode*) que, por um controle de fluxo (*ControlFlow*) transita para um nó que cria uma instância (*CreateObjectAction*). E por fim há uma transição de um fluxo de controle (*ControlFlow*) para o nó que indica o fim da atividade (*ActivityFinalNode*).

Para adicionar um diagrama de classe no Eclipse Modeling é necessário selecionar um novo diagrama de classe, posteriormente pode-se adicionar propriedades que são os atributos e as operações que são os métodos. Para adicionar um diagrama de atividade a cada método é necessário adicioná-lo selecionado o método em sua propriedade, adicionando um elemento comportamental de atividade ao mesmo. Posteriormente adicionar um diagrama de atividade a esse elemento, e por fim efetuar a modelagem dos nós e aretas necessários para a elaboração da funcionalidade

desejada.

Figura 19 – Diagrama de atividade createObject



Fonte: Autor.

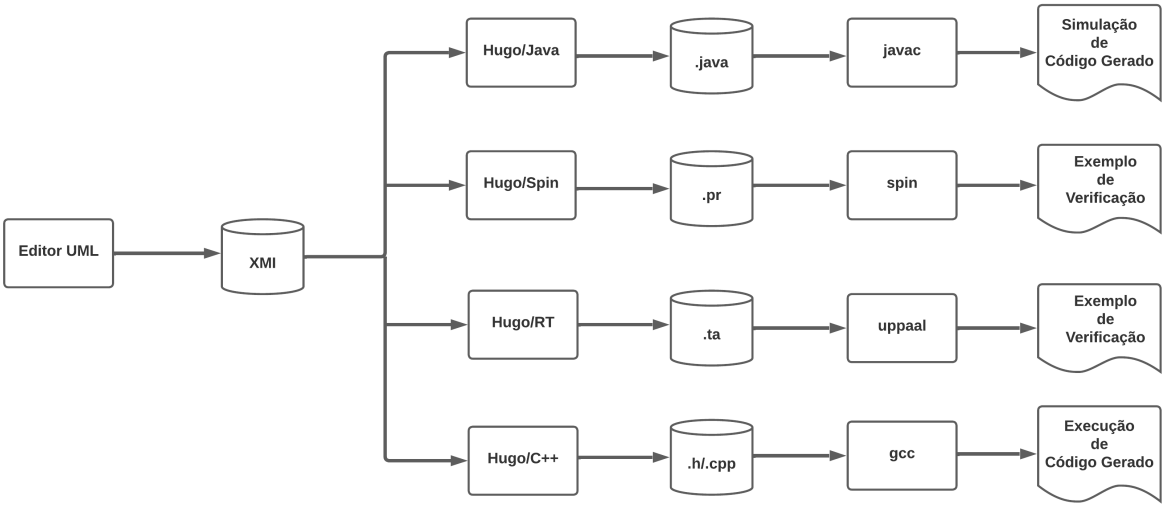
Todo projeto de modelagem de diagrama de atividade deve conter um nó inicial *InitialNode* e um nó final de atividade *ActivityFinalNode*. Uma restrição de escopo deste trabalho é não trabalhar com *ForkNode* e o *JoinNode*, ou seja, não trabalhar com tarefas em paralelo. Portanto, não suporta os nós de fim de fluxo, nó de junção, nó de bifurcação, região de expansão, nó de expansão e aresta de manipulação de exceção.

3.3 HUGO-RT

O Hugo-RT (KNAPP, 2022) é uma ferramenta projetada para gerar código e obter validação de modelos UML. Esses modelos UML podem conter classes, máquinas de estado, interações e restrições OCL, gerando assim código para a linguagem de sistema verificador de modelo em tempo real UPPAAL, verificador de modelo SPIN,

além de código Java e C++, em particular para o Arduino. Sua estrutura de arquitetura de ferramenta é desenvolvida na linguagem Java. A Figura 20 exemplifica melhor o funcionamento da ferramenta Hugo-RT.

Figura 20 – Visão geral do Hugo



Fonte: Modificado de Merz e Knapp (2001).

Segundo (SCHäFER; KNAPP; MERZ, 2001), o Hugo-RT é uma ferramenta projetada para verificação de máquinas cronometradas de estados com modelagem UML. Deste modo, eles especificam que o estado atual de uma máquina de estados é dada pela configuração ativa e pelo conteúdo de sua fila de eventos. Ainda segundo os autores, a fila de eventos contém os eventos que não foram tratados pela máquina, e os estados ativos formam uma árvore de estados ativos.

Uma transição é realizada quando todos os estados de origem contidos na configuração de estado ativo, e se o evento tiver guarda verdadeira e corresponder ao gatilho atual. É caracterizada como uma transição consistente quando duas transições não compartilham um estado de origem. Por fim o estado mais baixo contém todos os estados de origem, sendo desativos, depois as ações de transição são executadas e seus estados de destino ativado.

3.4 MÉTODO

Após realizar a modelagem dos diagramas comportamentais de atividade para classe, é compilado a ferramenta Hugo-RT (KNAPP, 2022) informando o endereço do diretório em que está contido o arquivo *.uml*. Posteriormente é realizado a leitura do arquivo *.uml*, onde é realizado a priori a procura do nó inicial. Em seguida há execução de um laço *while* cuja condição é chegar até o nó de atividade final. Enquanto o nó de atividade final não é encontrado é gerado código para cada respectivo nó de atividade,

seguindo o fluxo de controle, definido segundo a modelagem, para cada diagrama de atividade haverá um nó inicial e um nó de atividade final.

Se houver algum nó de decisão este deve ser empilhado, consecutivamente todos os nós de decisão serão empilhados em um variável local. Segue-se a sequência dos nós de controle, estipulada pelas suas dependências, até que o próximo nó atinja um nó de junção. Quando um nó de junção é encontrado o nó de decisão no topo da pilha é retirado, e a sequência do nó de controle é mudada de forma que às duas arestas que provêm do nó de decisão tenham seus próximos nós verificados e a geração de código realizada. É possível verificar no Apêndice A e no algoritimo 1.

Algorithm 1 Logic for decision node

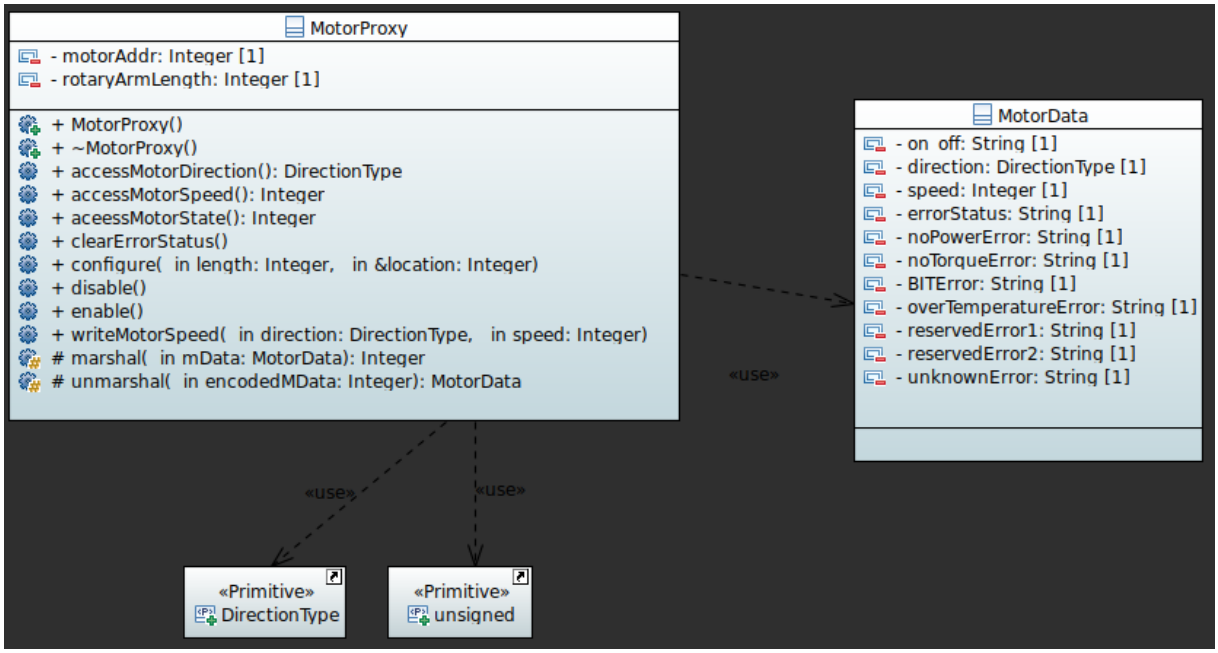
```
1: while nextNode ≠ ActivityFinalNode do
2:     if nextNode == DecisionNode then
3:         stackDecisionNode ← addTop(nextNode)                ▷ Add to stack
4:     else if nextNode == MergeNode then
5:         if auxMergeNode != visited then
6:             stackDecisionNode ← removeTop()                 ▷ Remove from stack
7:             auxMergeNode ← addKeyword(visited)
8:             resultBuilderFull ← doTranslation(nextNode))    ▷ Call the code
   generation function
9:         end if
10:    end if
11:    resultBuilderFull ← doTranslation(nextNode))            ▷ Call the code generation
   function
12: end while
```

Sendo assim, há uma estrutura de programação estipulada para cada nó de atividade proveniente da modelagem. Para cada há uma abordagem para acessar uma característica específica. Através de um diagrama de classe obtido como base para modelagem, por meio do livro de padrões de projeto para sistemas embarcados em C (DOUGLASS, 2011). A Figura 7 representa um exemplo utilizado para geração de código, esse exemplo traz um padrão de proxy de hardware que utiliza uma classe para encapsular qualquer acesso a um dispositivo de hardware, independentemente de sua interface física (DOUGLASS, 2011). Efetuando a modelagem no Eclipse Papyrus Modeling obtém-se a modelagem representada pela Figura 21.

Outros panoramas das modelagens podem ser observados nas Figuras 22, 24, 26, 28, 30, 32, 34, 36 e 38. Estas figuras retratam atividades provenientes dos métodos da classe *MotorProxy* na Figura 21. A Figura 22 retrata a atividade *accessMotorDirection*, a Figura 24 demostra a atividade *accessMotorSpeed*, a Figura 26 representa a atividade *accessMotorState*, a Figura 28 retrata a atividade *clearErrorStatus*, a Figura 30 simboliza a atividade *configure*, já a Figura 32 descreve

a atividade *disable*, a Figura 34 expressa a atividade *writeMotorSpeed*, a Figura 36 simboliza a atividade *marshal* e a Figura 38 demonstra a atividade *unmarshal*.

Figura 21 – Diagrama de classe MotorProxy com Eclipse Papyrus Modeling



Fonte: Autor.

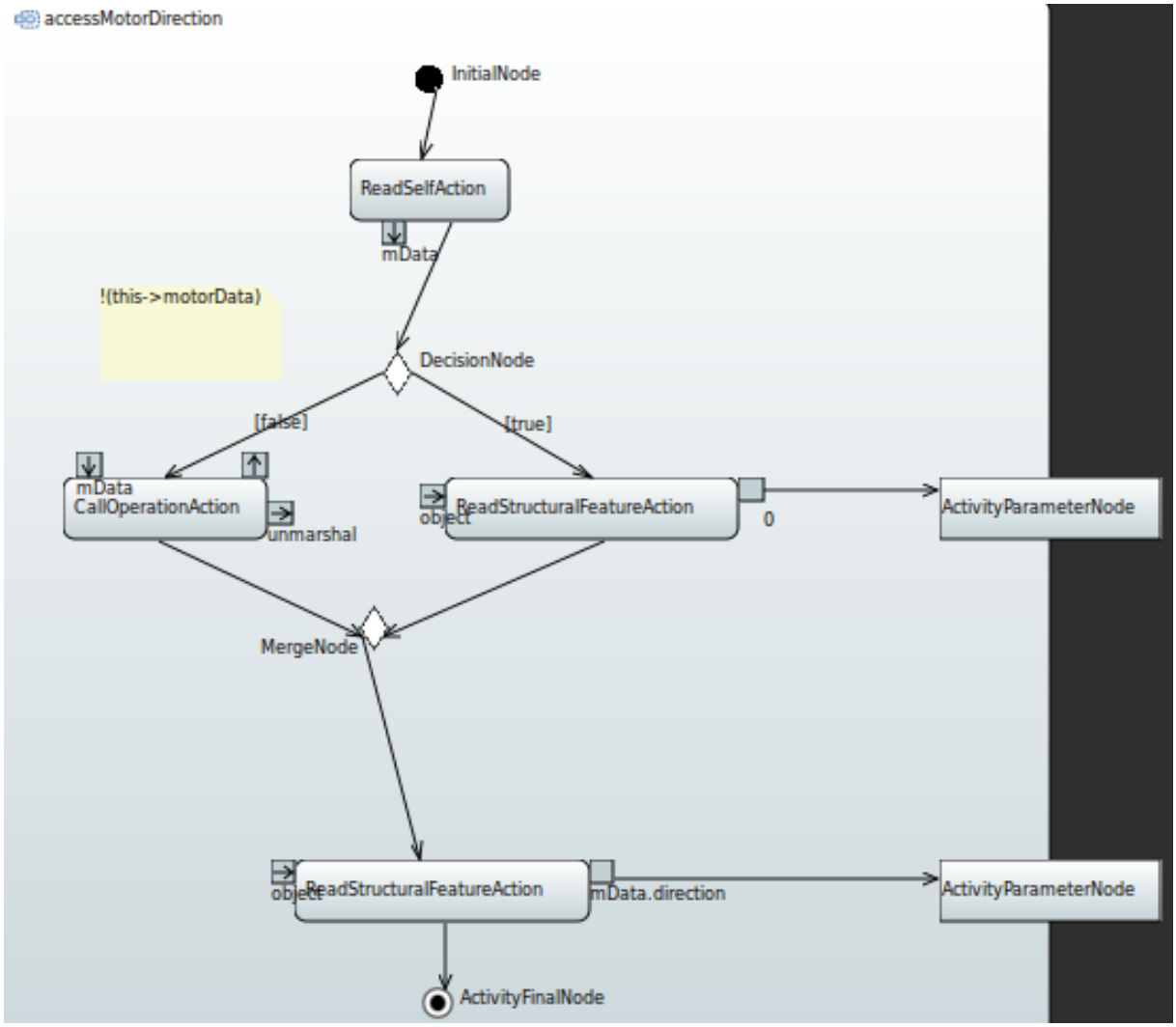
De modo geral os comportamentos das ações foram padronizados para obter uma saída coerente com a linguagem C++. A Figura 22 pode ser usada como exemplo para demonstrar o que cada ação representa em código em C++. A ação *ReadSelfAction* cria uma variável do tipo *auto* com o nome determinado no pino de resultado, e associa o ponteiro *this* que representa o valor de um atributo da classe. Já o nó *DecisionNode* gera o fragmento de código que descreve a instrução *if*, sendo preciso criar uma decisão de entrada do tipo *opaqueBehavior*. Necessário estipular a linguagem de programação e o trecho de código que descreve a condição do laço *if*. A criação do nó *DecisionNode* gera um nó de comentário que demonstra a condição contida na declaração quando atribuído o nome da decisão ao conteúdo da mesma.

O *MergeNode* gera o fragmento de código caracterizado pela condição *else*, ou seja, se a instrução dentro do *if* for falsa realiza a condição *else*. A ação *CallOperationAction* atua como uma chamada de um método, o nome do objeto deve estar no pino de alvo e o método deve ser especificado no pino de operação. Outra ação que realiza o retorno de objetos é a *ReadStructuralFeatureAction*, sendo preciso nomear o objeto a ser retornado através do pino de resultado. Essa ação deve ser ligada a ação *ActivityParameterNode* por meio do fluxo de objeto para indicar que está acontecendo uma saída de um objeto do diagrama de atividade.

Por fim a Figura 22 traz consigo os nós de início e fim de atividade, respectivamente, *InitialNode* e *ActivityFinalNode*. Esses nós não criam código no

processo de geração, e são utilizados somente para controle de onde inciar e finalizar o processo.

Figura 22 – Diagrama de atividade accessMotorDirection com Eclipse Papyrus Modeling



Fonte: Autor.

A seguir serão demonstrados as modelagens em diagramas de atividade realizadas para cada método e seus respectivos códigos gerados pela ferramenta Hugo-RT, onde diagrama de atividade *accessMotorDirection* demonstrado pela Figura 22, após passar pelo gerador de código Hugo-RT resulta no código representado pela Figura 23. Como o livro (DOUGLASS, 2011) foi tido como base para a modelagem, e o mesmo traz consigo código em C, se faz necessário uma interpretação do código para a linguagem de código gerado C++.

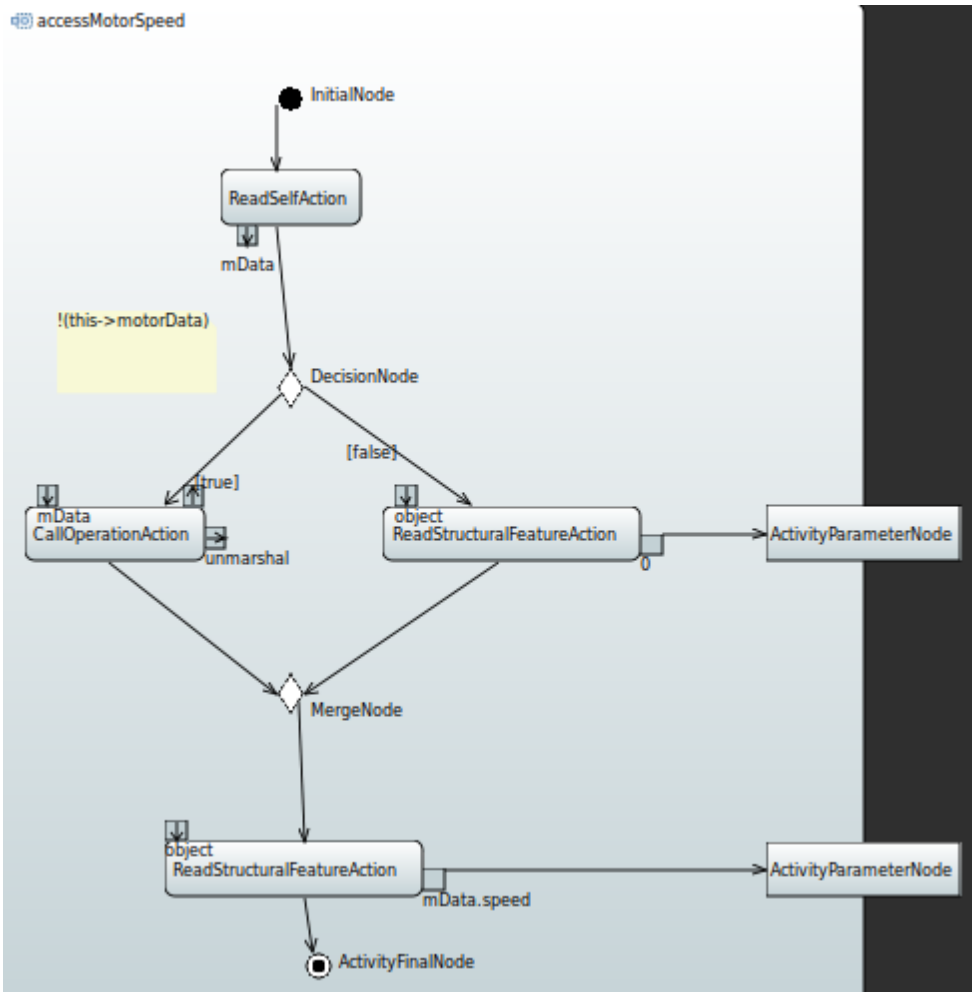
Figura 23 – Código gerado do diagrama de atividade accessMotorDirection

```
void MotorProxy::accessMotorDirection() {
  auto mData = this;
  if (!(this->motorData)) {
  return 0;} else {
  mData.unmarshal();

  }
  return mData.direction;
}
```

Fonte: Autor.

Já o diagrama de atividade *accessMotorSpeed* exposta pela Figura 24, após passar pelo gerador de código Hugo-RT resulta no código ilustrado pela Figura 25.

Figura 24 – Diagrama de atividade accessMotorSpeed com Eclipse Papyrus Modeling



Fonte: Autor.

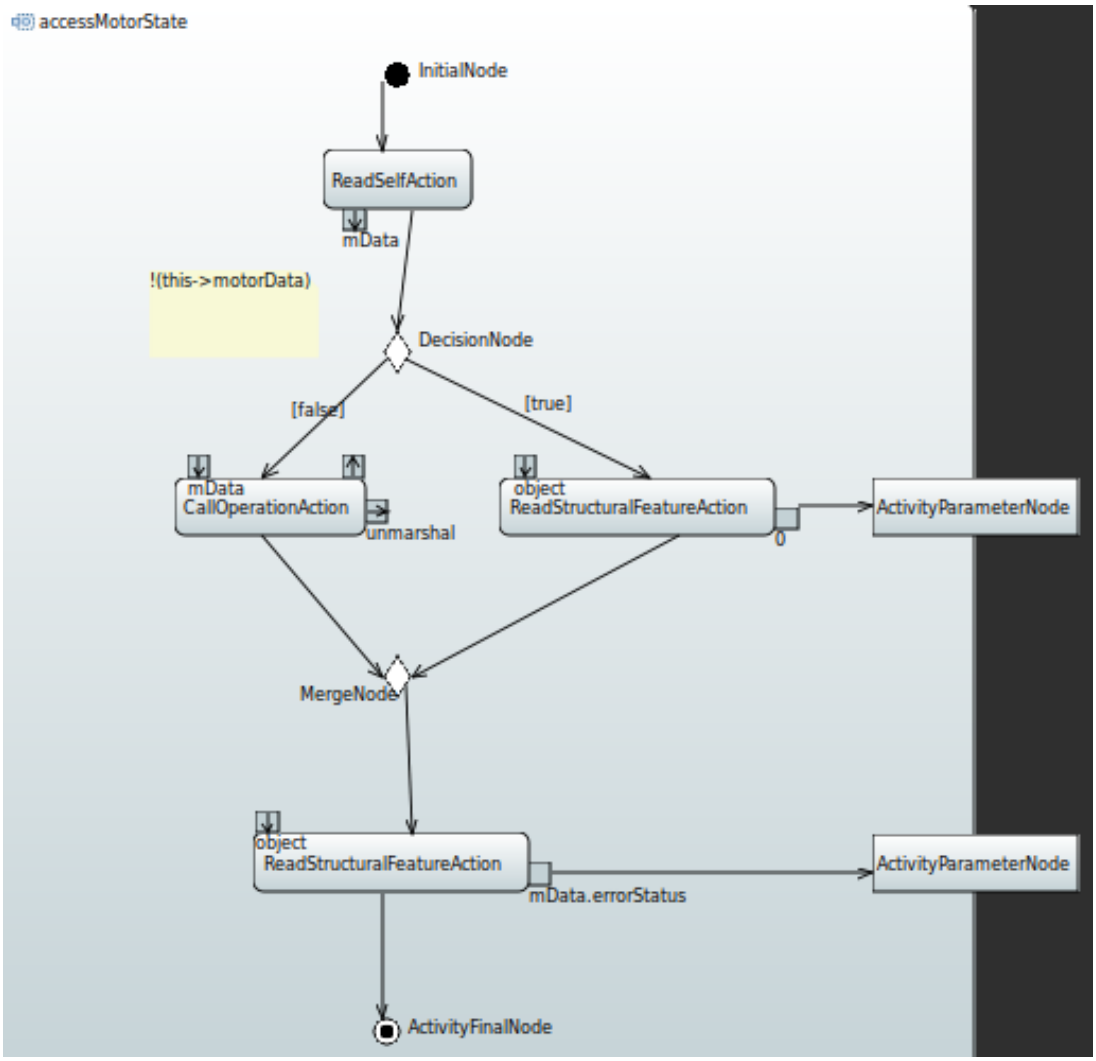
Figura 25 – Código gerado do diagrama de atividade accessMotorSpeed

```
void MotorProxy::accessMotorSpeed() {
  auto mData = this;
  if (!(this->motorData)) {
  return 0;} else {
  mData.unmarshal();

  }
  return mData.speed;
}
```

Fonte: Autor.

O diagrama de atividade *accessMotorState* evidenciado pela Figura 26, após passar pelo gerador de código Hugo-RT resulta no código demostrado pela Figura 27.

Figura 26 – Diagrama de atividade accessMotorState com Eclipse Papyrus Modeling



Fonte: Autor.

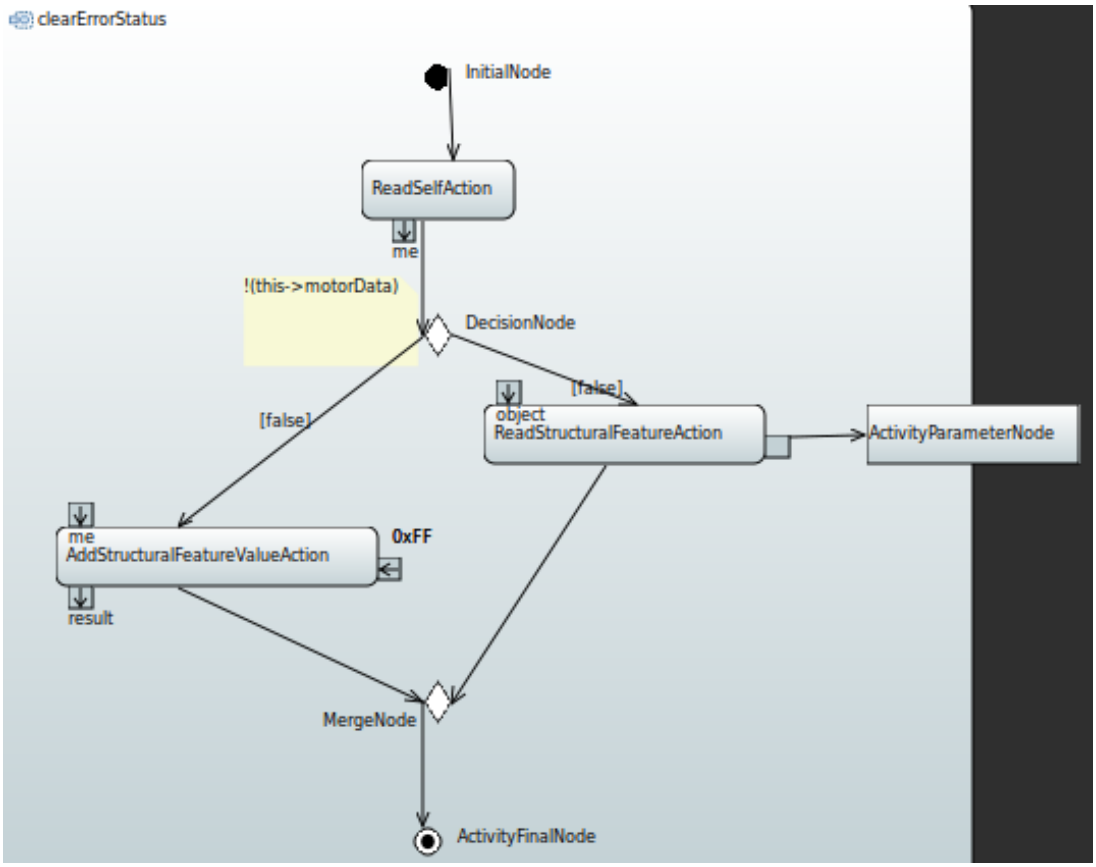
Figura 27 – Código gerado do diagrama de atividade accessMotorState

```
void MotorProxy::aceessMotorState() {
  auto mData = this;
  if (!(this->motorData)) {
  return 0;} else {
  mData.unmarshal();

  }
  return mData.errorStatus;
}
```

Fonte: Autor.

A atividade *clearErrorStatus* representada pela Figura 28, após passar pelo gerador de código Hugo-RT resulta no código exposto pela Figura 29.

Figura 28 – Diagrama de atividade clearErrorStatus com Eclipse Papyrus Modeling



Fonte: Autor.

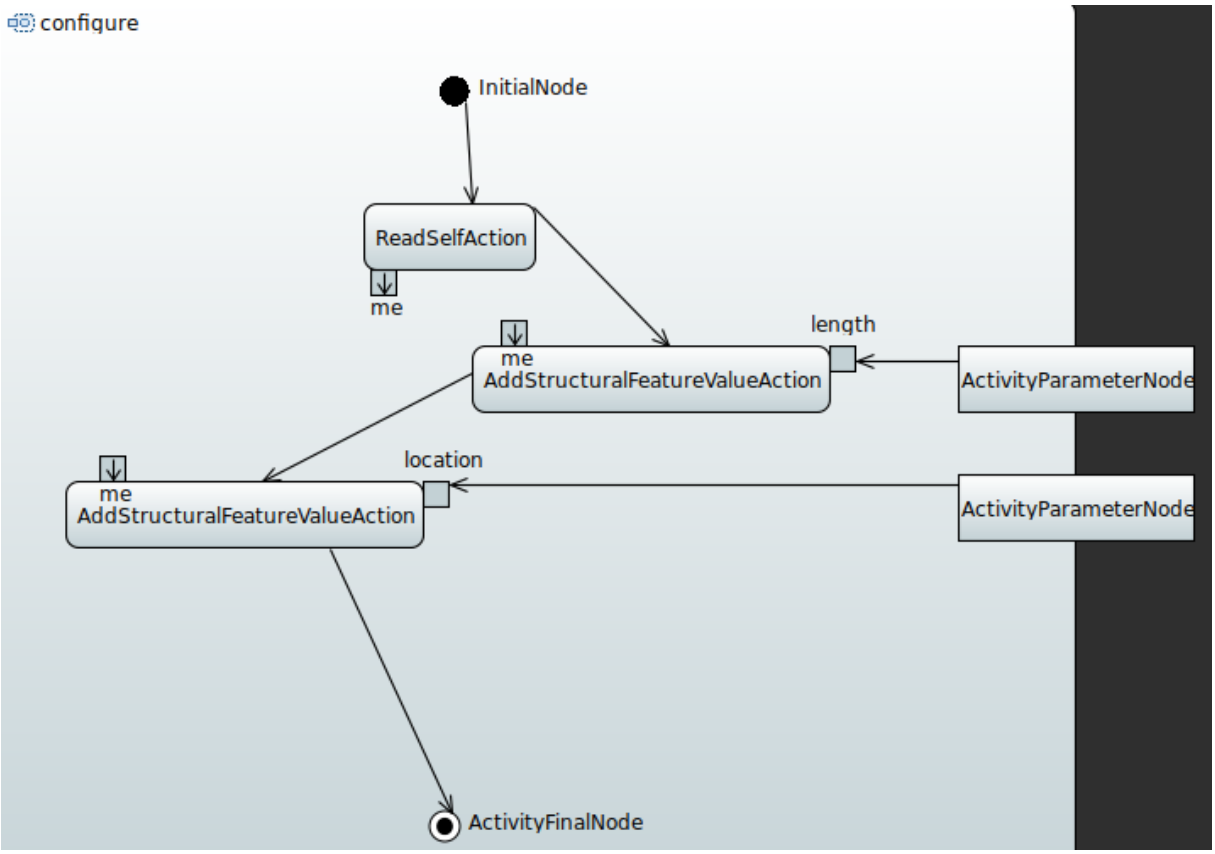
Figura 29 – Código gerado do diagrama de atividade clearErrorStatus

```
void MotorProxy::clearErrorStatus() {
  auto me = this;
  if (!(this->motorData)) {
  return  ;} else {
  me->motorAddr = 0xFF;

  }
}
```

Fonte: Autor.

O diagrama de atividade *configure* apresentado pela Figura 30, após passar pelo gerador de código Hugo-RT resulta no código ilustrado pela Figura 31.

Figura 30 – Diagrama de atividade configure com Eclipse Papyrus Modeling



Fonte: Autor.

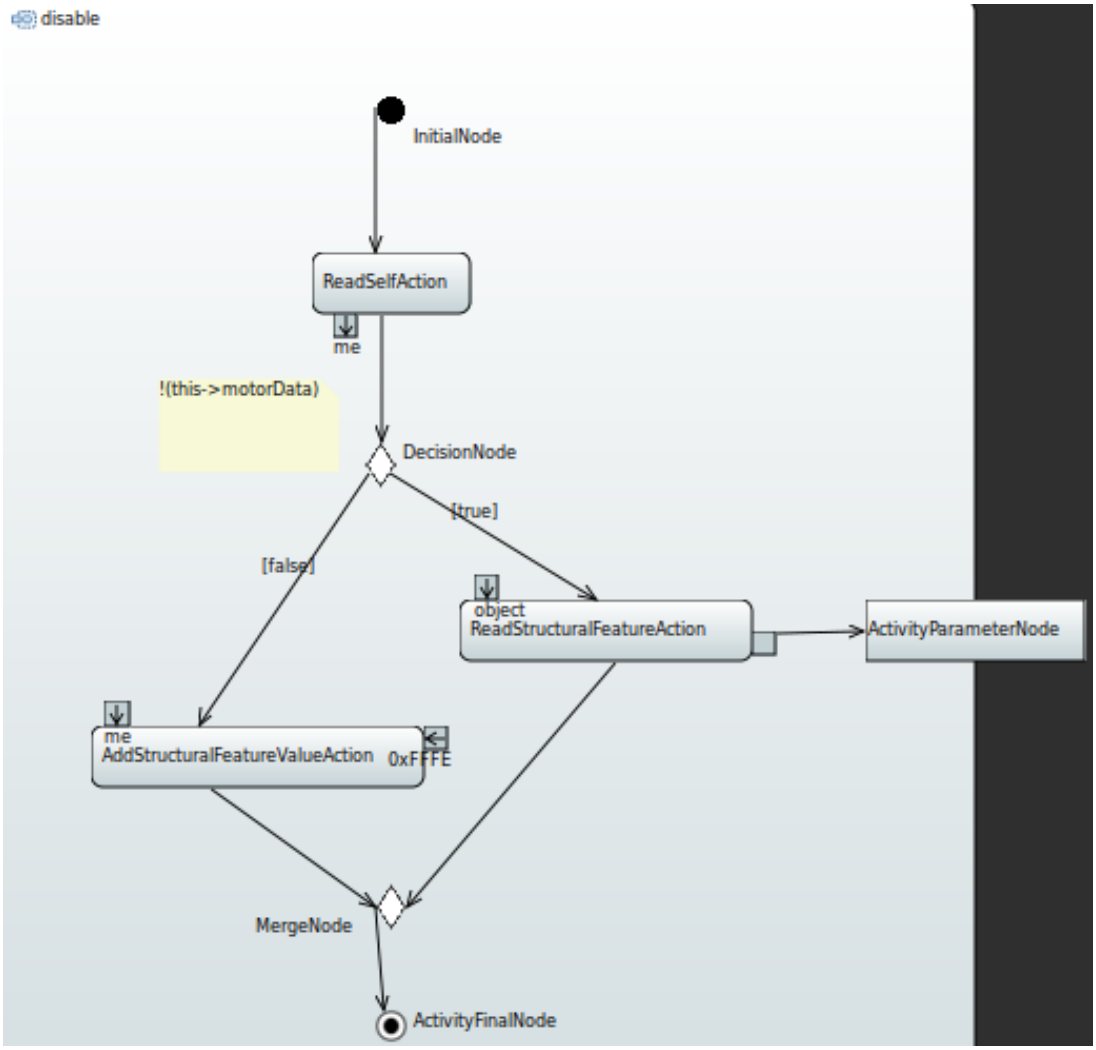
Figura 31 – Código gerado do diagrama de atividade configure

```
void MotorProxy::configure(int length, int _location) {
  auto me = this;
  me->rotaryArmLength = length;
  me->motorAddr = location;
}
```

Fonte: Autor.

A atividade *disable* exposta pela Figura 32, após passar pelo gerador de código Hugo-RT resulta no código ilustrado pela Figura 33.

Figura 32 – Diagrama de atividade disable com Eclipse Papyrus Modeling



Fonte: Autor.

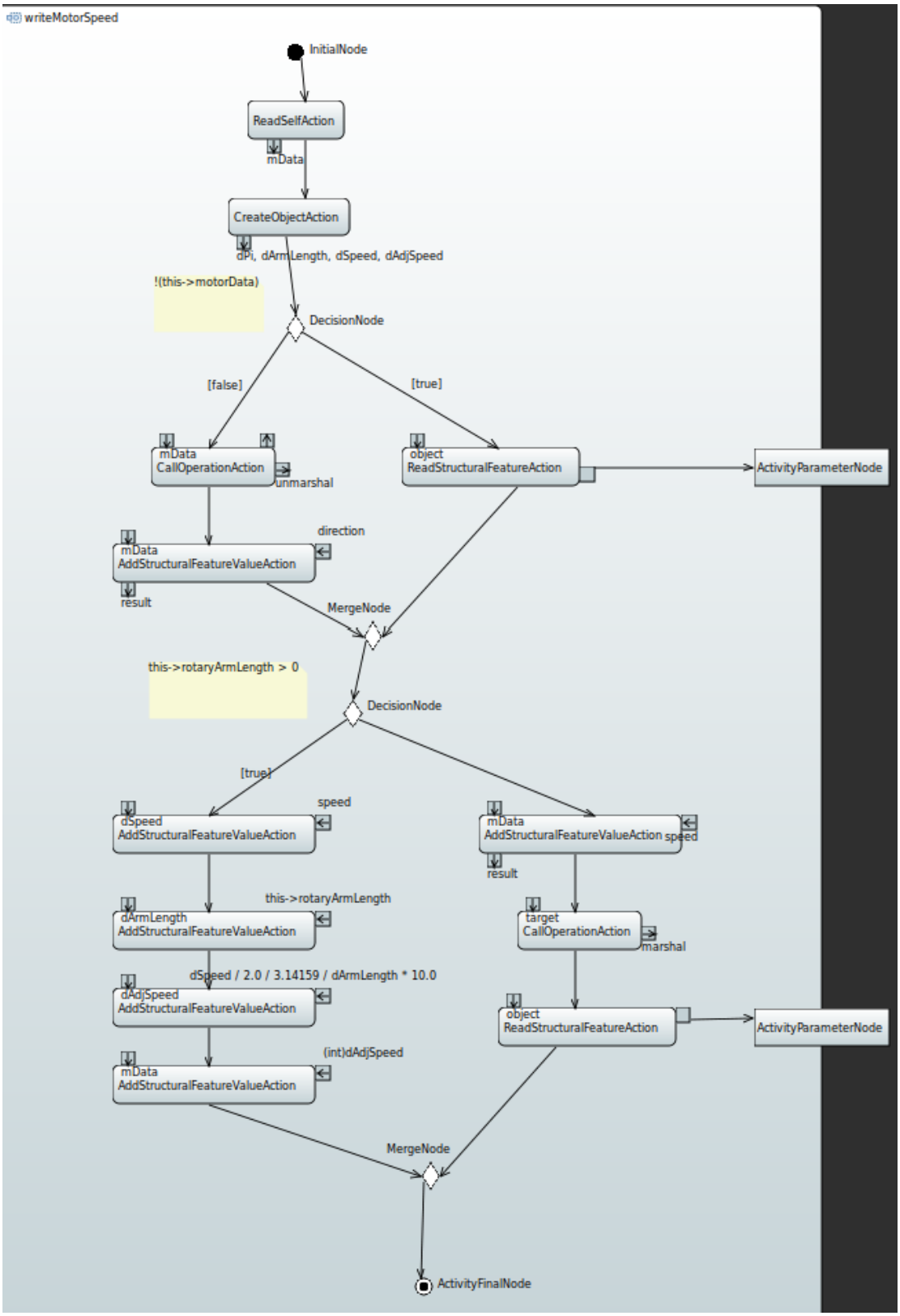
Figura 33 – Código gerado diagrama de atividade disable

```
void MotorProxy::disable() {
  auto me = this;
  if (!(this->motorData)) {
  return  ;} else {
  me->motorAddr = 0xFFFE;

  }
}
```

Fonte: Autor.

Já a atividade *writeMotorSpeed* interpretado pela Figura 34, após passar pelo gerador de código Hugo-RT resulta no código expressado pela Figura 35.

Figura 34 – Diagrama de atividade writeMotorSpeed com Eclipse Papyrus Modeling



Fonte: Autor.

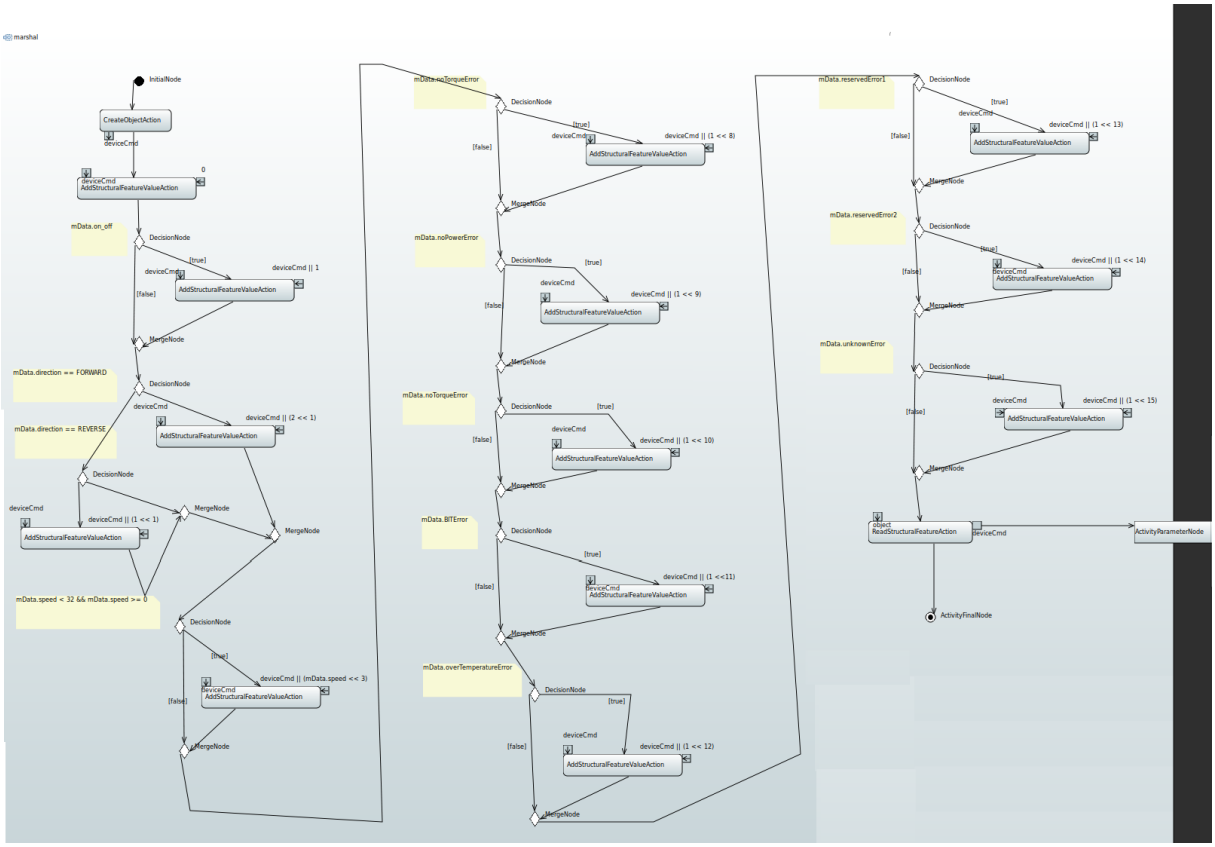
Figura 35 – Código gerado diagrama de atividade writeMotorSpeed

```
void MotorProxy::writeMotorSpeed( direction, int speed) {
  auto mData = this;
  double dPi, dArmLength, dSpeed, dAdjSpeed;
  if (!(this->motorData)) {
  return  ;} else {
  mData.unmarshal();
  mData->direction = direction;

  }
  if (this->rotaryArmLength > 0) {
  dSpeed = speed;
  dArmLength = this->rotaryArmLength;
  dAdjSpeed = dSpeed / 2.0 / 3.14159 / dArmLength * 10.0;
  mData->speed = (int)dAdjSpeed;
  } else {
  mData->speed = speed;
  target.marshal();
  return  ;
  }
}
```

Fonte: Autor.

O diagrama de atividade *marshal* demostrado pela Figura 36, após passar pelo gerador de código Hugo-RT resulta no código expressado pela Figura 37.

Figura 36 – Diagrama de atividade marshal com Eclipse Papyrus Modeling



Fonte: Autor.

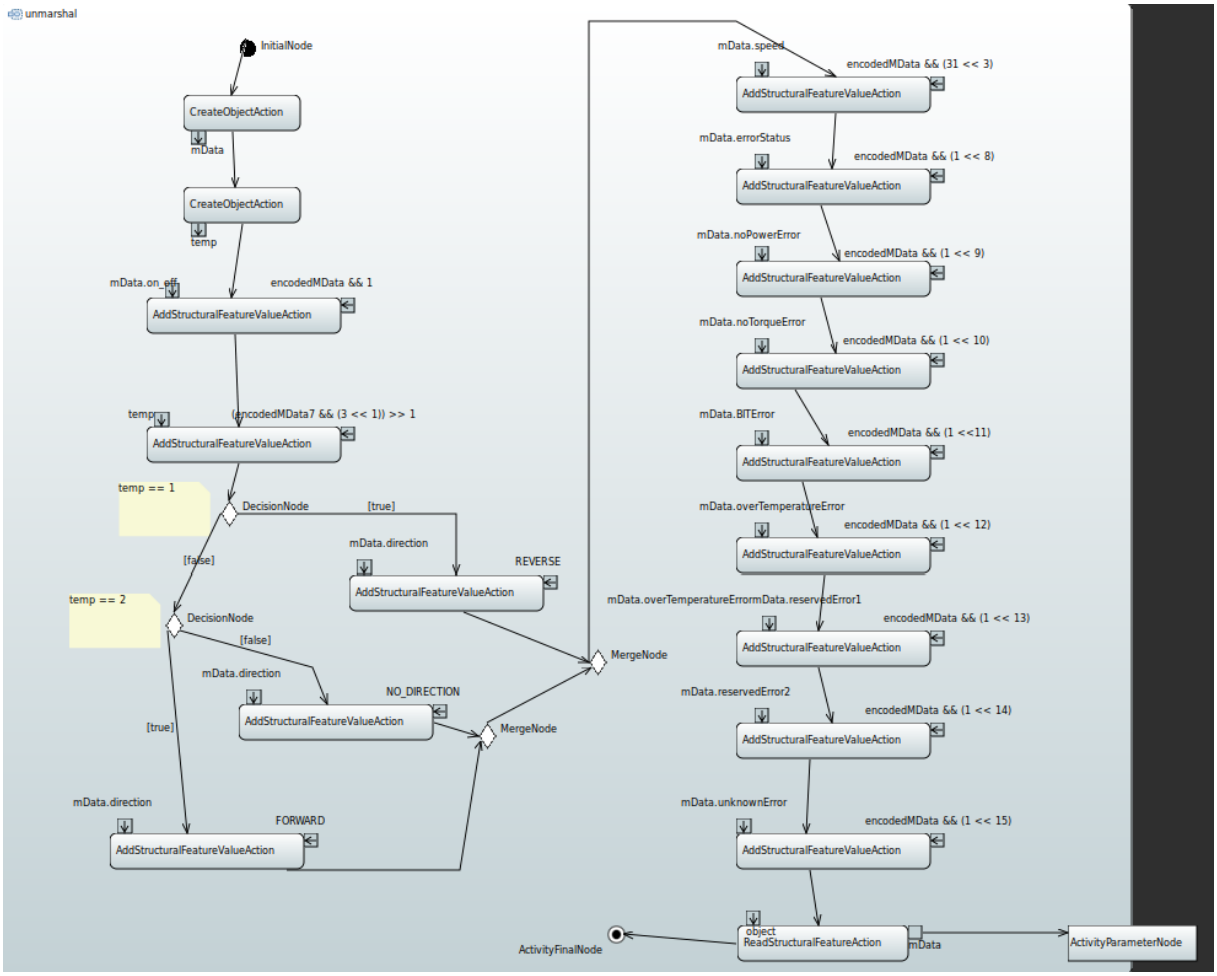
Figura 37 – Código gerado do diagrama de atividade marshal

```
void MotorProxy::marshal(MotorData mData) {
  int deviceCmd;
  deviceCmd = 0;
  if (mData.on_off) {
  deviceCmd = deviceCmd || 1;

  }
  if (mData.direction == FORWARD) {
  deviceCmd = deviceCmd || (2 << 1);
  } else {
  if (mData.direction == REVERSE) {
  deviceCmd = deviceCmd || (1 << 1);

  }

  }
  if (mData.speed < 32 && mData.speed >= 0) {
  deviceCmd = deviceCmd || (mData.speed << 3);

  }
  if (mData.errorStatus) {
  deviceCmd = deviceCmd || (1 << 8);

  }
  if (mData.noPowerError) {
  deviceCmd = deviceCmd || (1 << 9);

  }
  if (mData.noTorqueError) {
  deviceCmd = deviceCmd || (1 << 10);

  }
  if (mData.BITError) {
  deviceCmd = deviceCmd || (1 <<11);

  }
  if (mData.overTemperatureError) {
  deviceCmd = deviceCmd || (1 << 12);

  }
  if (mData.reservedError1) {
  deviceCmd = deviceCmd || (1 << 13);

  }
  if (mData.reservedError2) {
  deviceCmd = deviceCmd || (1 << 14);

  }
  if (mData.unknownError) {
  deviceCmd = deviceCmd || (1 << 15);

  }
  return deviceCmd;
}
```

Fonte: Autor.

A atividade *unmarshal* demonstrada pela Figura 38, após passar pelo gerador de código Hugo-RT resulta no código representado pela Figura 39.

Figura 38 – Diagrama de atividade unmarshal com Eclipse Papyrus Modeling



Fonte: Autor.

Figura 39 – Código gerado do diagrama de atividade unmarshal

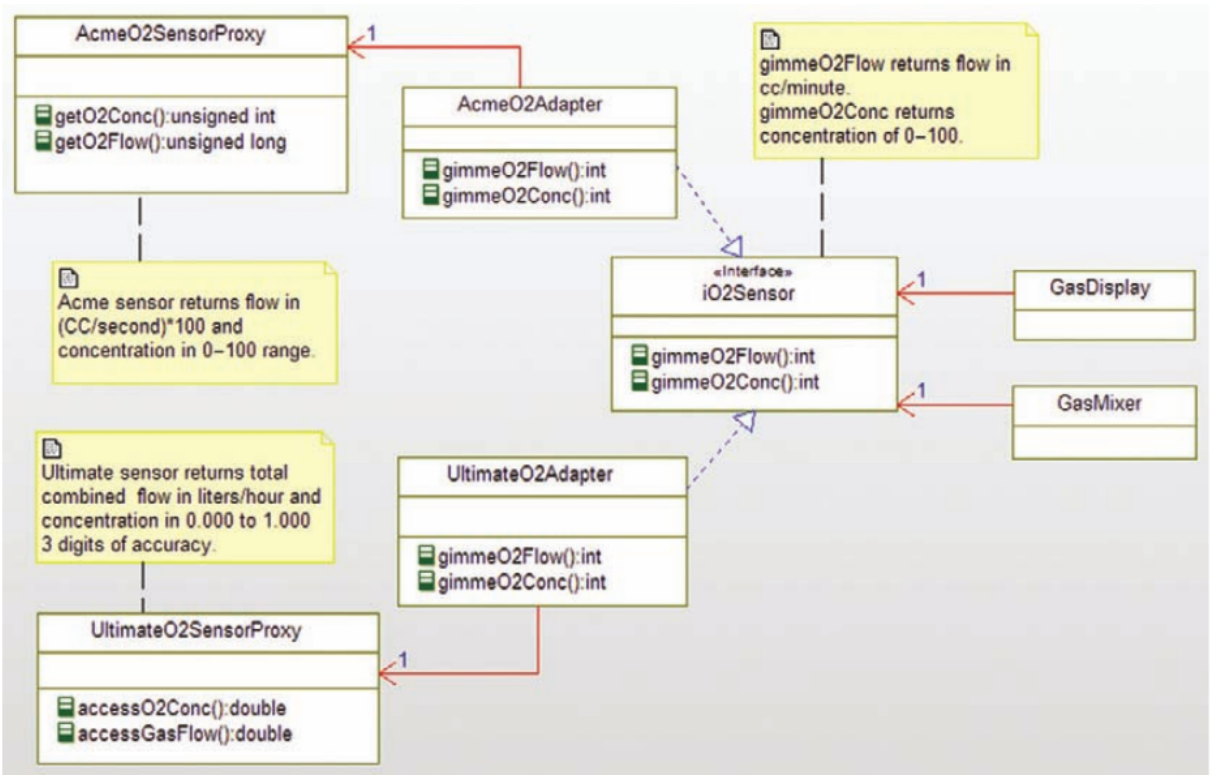
```
void MotorProxy::unmarshal(int encodedMData) {
  MotorData mData;
  int temp;
  mData.on_off = encodedMData && 1;
  temp = (encodedMData7 && (3 << 1)) >> 1;
  if (temp == 1) {
  mData.direction = REVERSE;
  } else {
  if (temp == 2) {
  mData.direction = FORWARD;
  } else {
  mData.direction = NO_DIRECTION;

  }

  }
  mData.speed = encodedMData && (31 << 3);
  mData.errorStatus = encodedMData && (1 << 8);
  mData.noPowerError = encodedMData && (1 << 9);
  mData.noTorqueError = encodedMData && (1 << 10);
  mData.BITError = encodedMData && (1 <<11);
  mData.overTemperatureError = encodedMData && (1 << 12);
  mData.overTemperatureErrormData.reservedError1 = encodedMData && (1 << 13);
  mData.reservedError2 = encodedMData && (1 << 14);
  mData.unknownError = encodedMData && (1 << 15);
  return mData;
}
```

Fonte: Autor.

Outro exemplo retirado do livro de padrões de projeto para sistemas embarcados em C (DOUGLASS, 2011), com finalidade de demonstrar a modelagem pode ser observado na Figura 40, e esse exemplo traz um padrão de adaptador de hardware. O padrão de adaptador de hardware é uma maneira de adaptar uma interface de hardware existente, ou seja, quando um aplicativo requisita ou usa uma interface e há a troca da interface para o hardware real fornecido (DOUGLASS, 2011).

Aplicando a modelagem no Eclipse Papyrus Modeling obtém-se o diagrama de classe Figura 41. Outras características importantes são retratadas nos diagramas de atividade da classe UltimateO2Adapter. A primeira ação cria um objeto por meio do *CreateObjectAction*, indicando o nome do objeto no pino de resultado. O objeto criado por meio *CreateObjectAction* resulta em um tipo automático que em C++ é defino como *auto*. Além dessa ação há o *AddStructuralFeatureValueAction* que atribui um valor para um objeto, podendo atribuir por um pino de entrada, que recebe o valor através do *ActivityParameterNode*, ou por um pino de valor que atribui diretamente por meio do nome do mesmo.

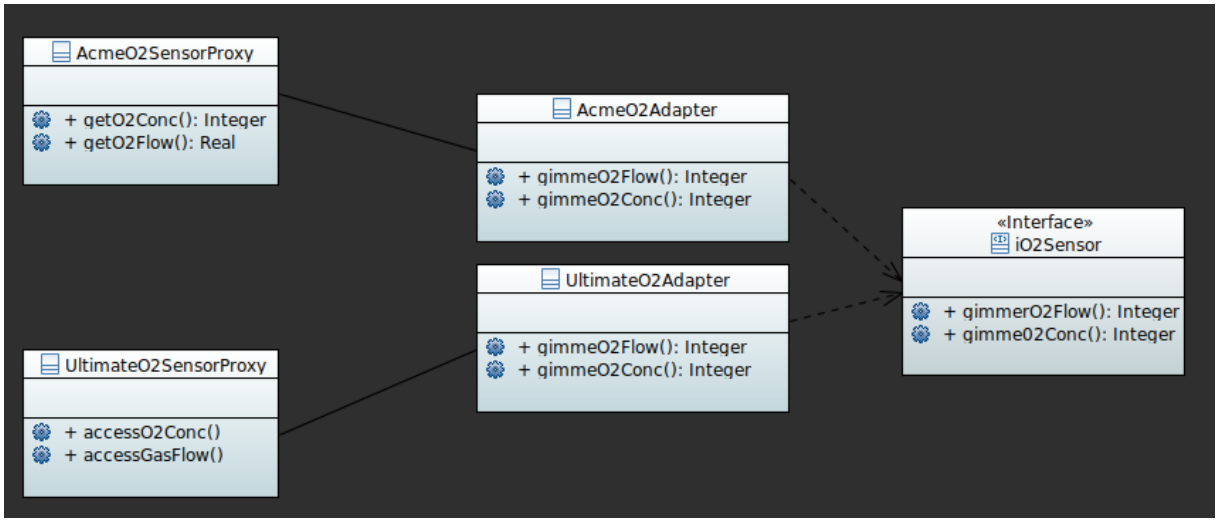
Figura 40 – Exemplo diagrama de classe de padrão de adaptador de Hardware



Fonte: Douglass (2011).

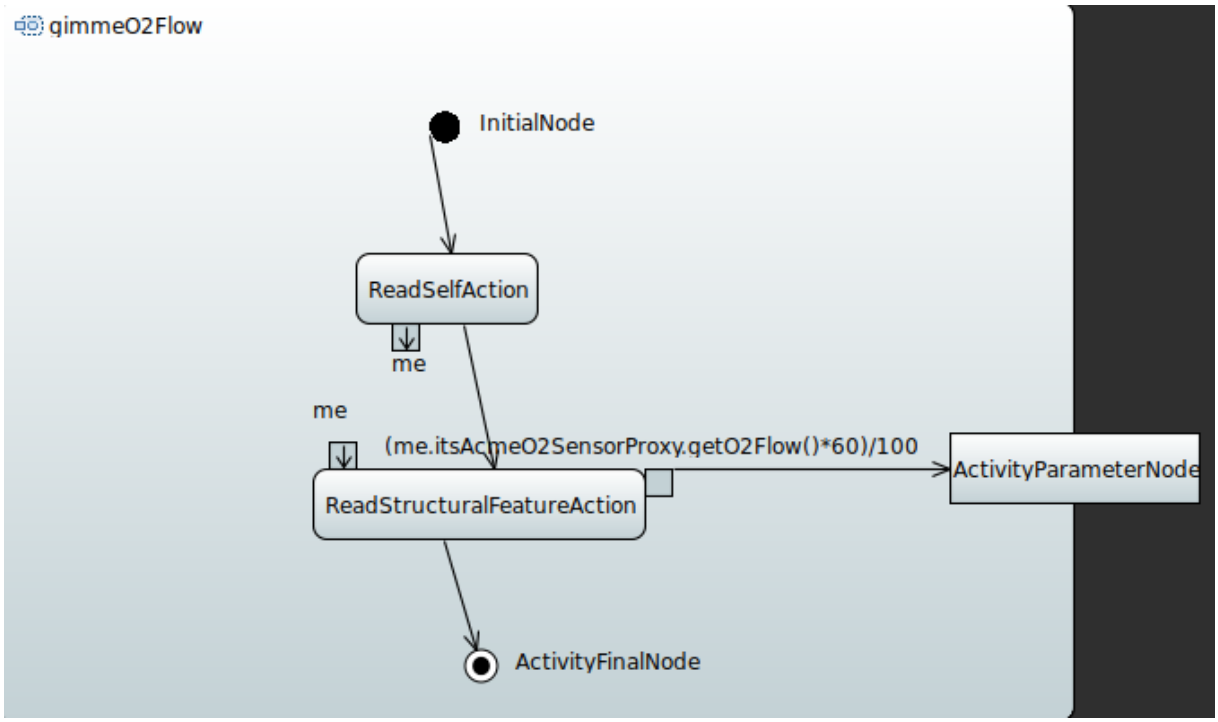
Os diagramas de atividade da classe *AcmeO2Adapter* podem ser visualizados nas Figuras 42 e 43, onde o código gerado para ambas podem ser observado na Figura 44. Já a classe *UltimateO2Adapter* contêm os diagramas de atividade demonstrados pelas Figuras 45 e 46, que geram o código demonstrado pela Figura 47.

Figura 41 – Diagrama de classe Adapter com Eclipse Papyrus Modeling



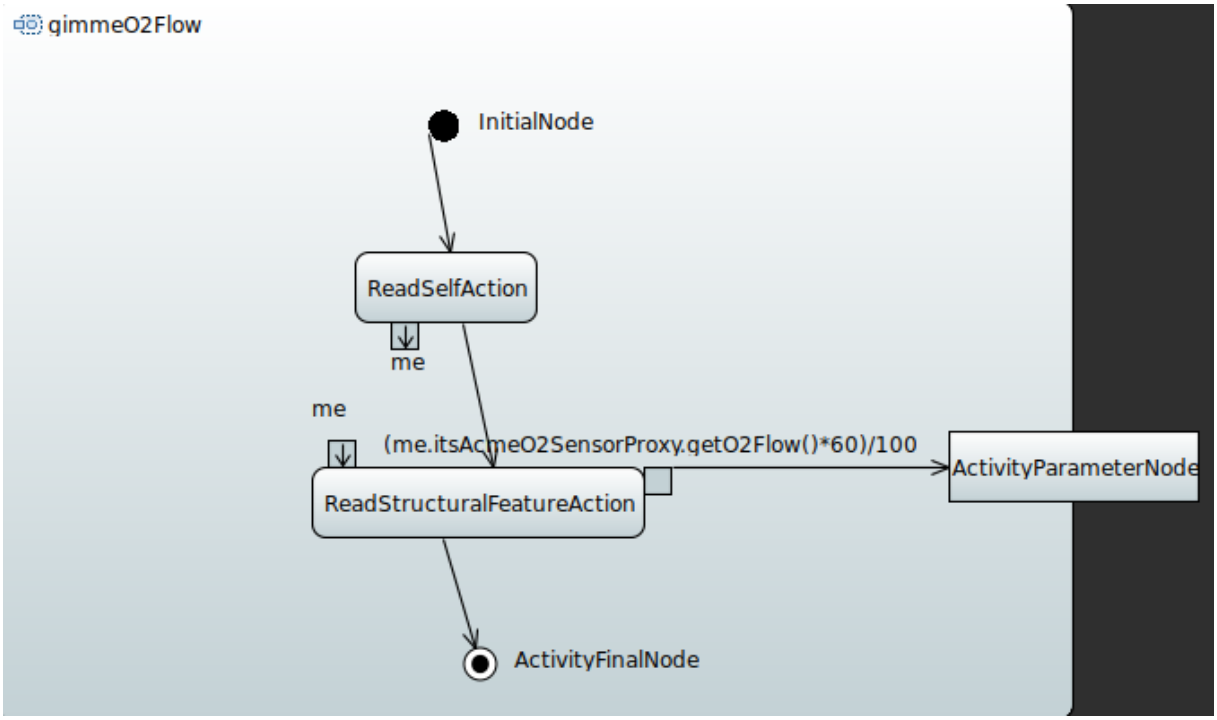
Fonte: Autor.

Figura 42 – Diagrama de atividade gimmeO2Flow da classe AcmeO2Adapter com Eclipse Papyrus Modeling



Fonte: Autor.

Figura 43 – Diagrama de atividade gimmeO2Conc da classe AcmeO2Adapter com Eclipse Papyrus Modeling



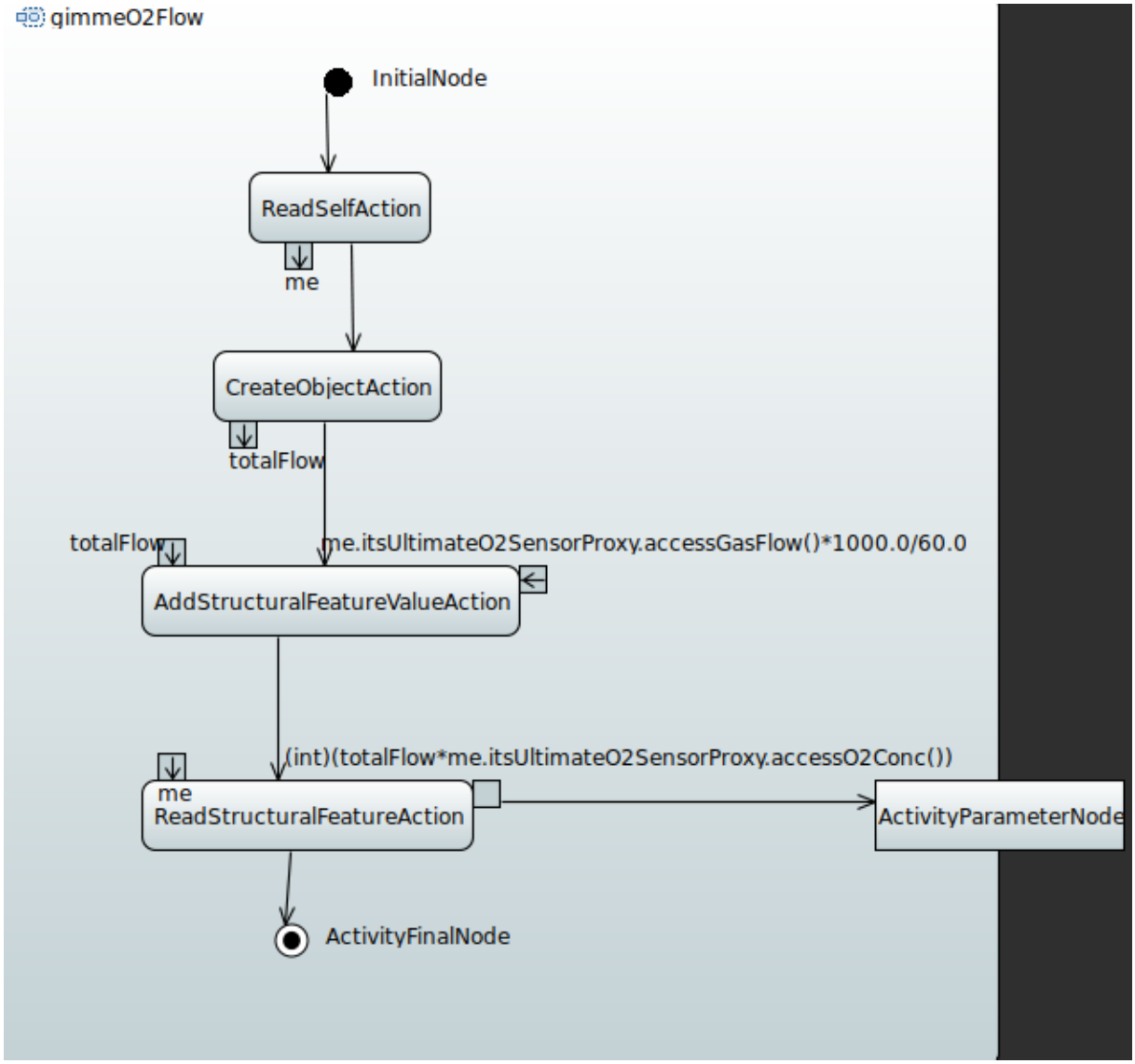
Fonte: Autor.

Figura 44 – Código gerado do diagrama de atividade gimmeO2Flow e gimmeO2Conc da classe AcmeO2Adapter

```
#include "../include/AcmeO2Adapter.hh"

AcmeO2Adapter::AcmeO2Adapter() {
}

void AcmeO2Adapter::gimmeO2Conc() {
  auto me = this;
  return (me.itsAcmeO2SensorProxy.getO2Flow()*60)/100;
}

void AcmeO2Adapter::gimmeO2Flow() {
  auto me = this;
  return me.itsAcmeO2SensorProxy.getO2Conc();
}
```

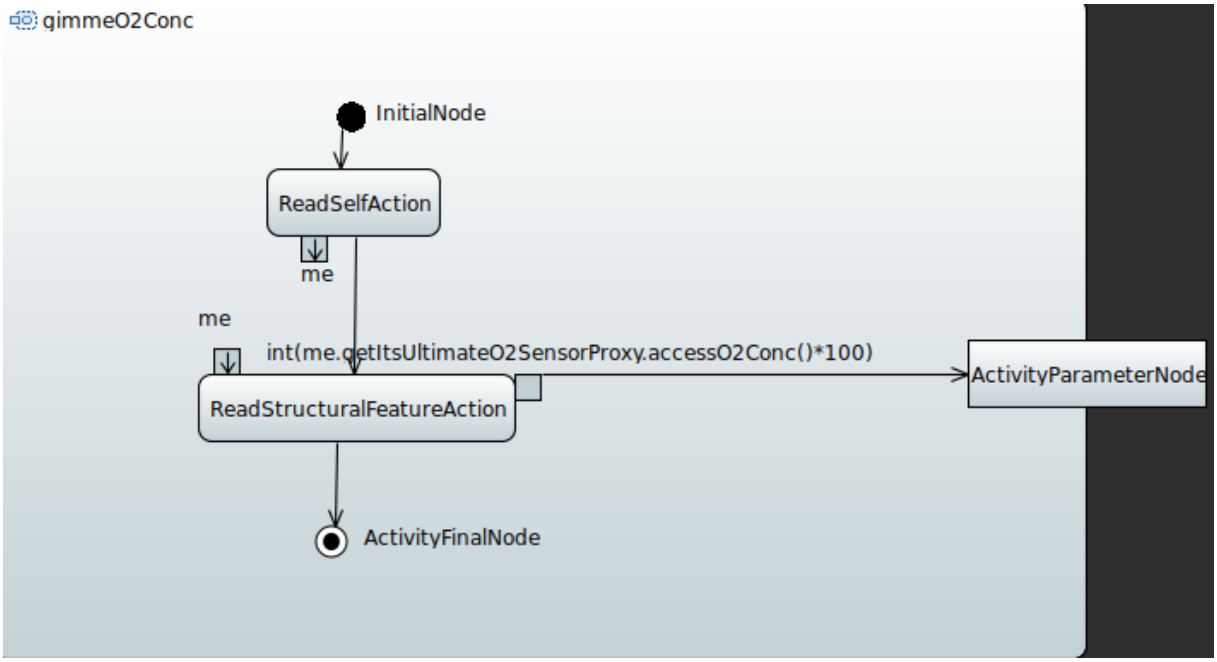
Fonte: Autor.

Figura 45 – Diagrama de atividade gimmeO2Flow da classe UltimateO2Adapter com Eclipse Papyrus Modeling



Fonte: Autor.

Figura 46 – Diagrama de atividade gimmeO2Conc da classe UltimateO2Adapter com Eclipse Papyrus Modeling



Fonte: Autor.

Figura 47 – Código gerado do diagrama de atividade gimmeO2Flow e gimmeO2Conc da classe UltimateO2Adapter

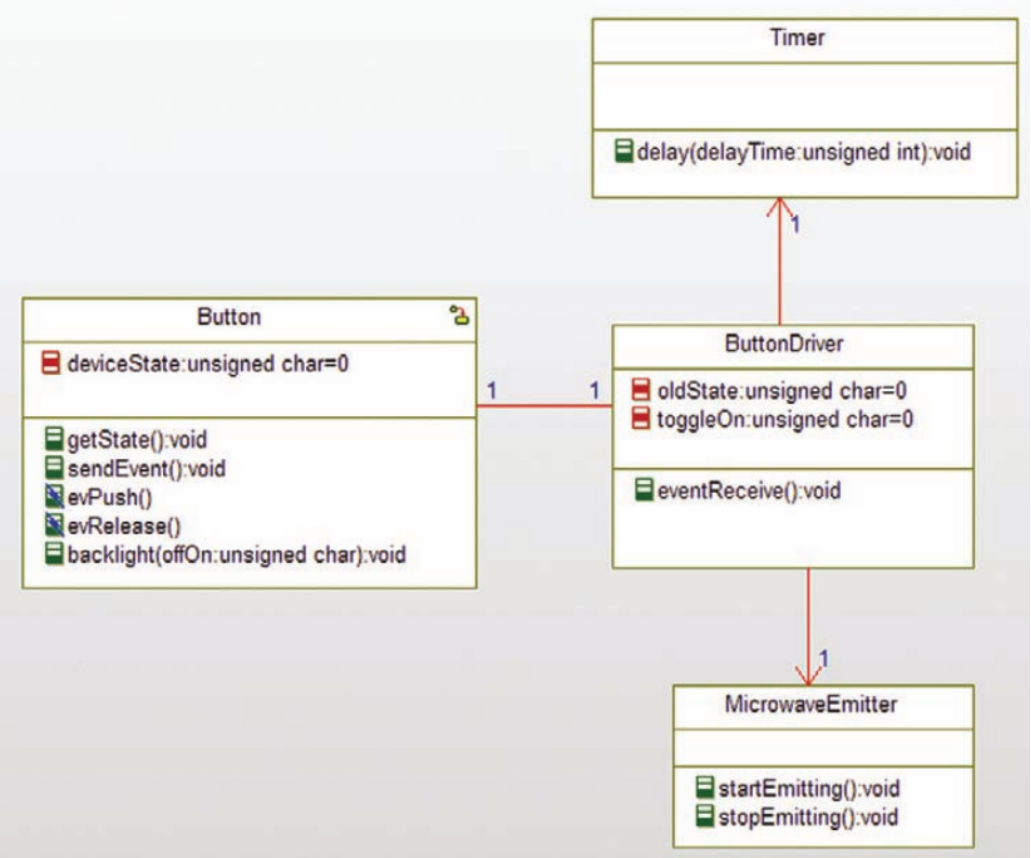
```
#include "../include/UltimateO2Adapter.hh"

UltimateO2Adapter::UltimateO2Adapter() {
}

void UltimateO2Adapter::setUltimateo2sensorproxy(UltimateO2SensorProxy* ultimateo2sensorproxy) {
  this->ultimateo2sensorproxy = ultimateo2sensorproxy;
}

UltimateO2SensorProxy* UltimateO2Adapter::getUltimateo2sensorproxy() {
  return this->ultimateo2sensorproxy;
}

void UltimateO2Adapter::gimmeO2Conc() {
  auto me = this;
  return int(me.getItsUltimateO2SensorProxy.accessO2Conc()*100);
}

void UltimateO2Adapter::gimmeO2Flow() {
  auto me = this;
  double totalFlow;
  totalFlow = me.itsUltimateO2SensorProxy.accessGasFlow()*1000.0/60.0;
  return (int)(totalFlow*me.itsUltimateO2SensorProxy.accessO2Conc());
}
```

Fonte: Autor.

4 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). Pode ser observado na Figura 48. O exemplo citado é utilizado para dispositivos de entrada de sistemas digitais como botões, interruptores e relés eletromecânicos, onde as conexões de metal trazem consigo uma taxa de erro provenientes da deformação do metal, podendo causar uma abundância de erros em razão da alta velocidade de resposta do sistema na ordem dos milissegundos. Para reduzir os vários sinais em um só é necessário esperar um período após o dispositivo ser acionado para verificar o sinal gerado (DOUGLASS, 2011). Aplicando a modelagem por meio da ferramenta Eclipse Papyrus Modeling obtém-se a Figura 49.

Figura 48 – Exemplo diagrama de classe de padrão de Debouncing

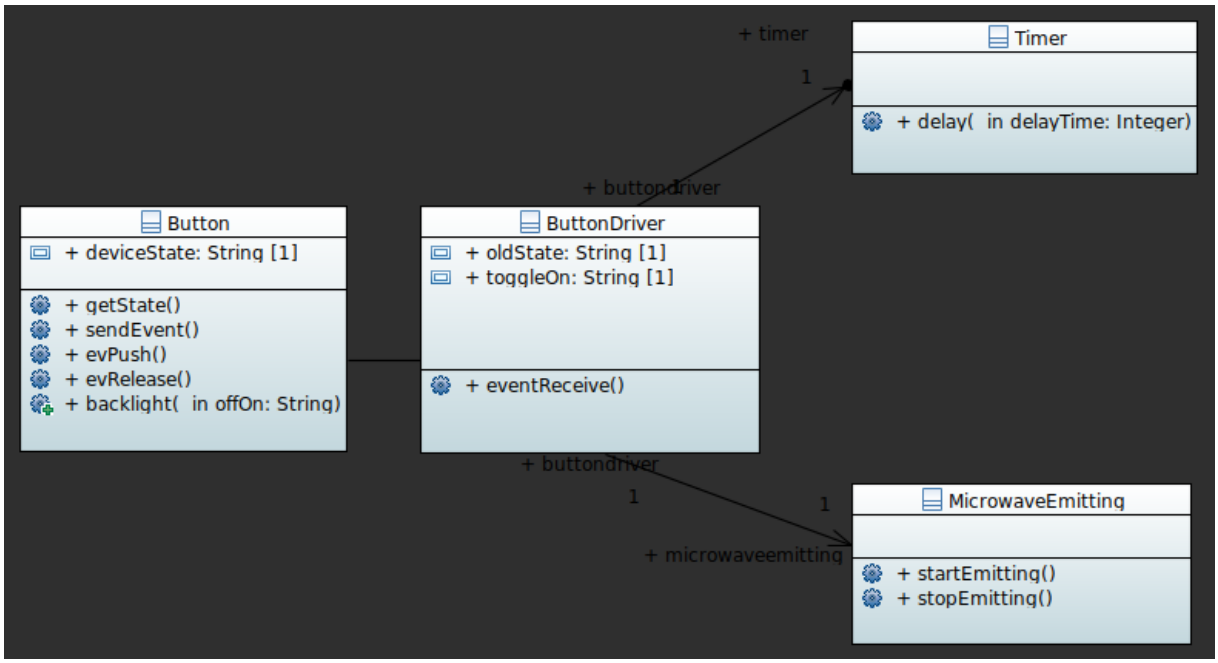


Fonte: Douglass (2011).

Foram implemntados dois métodos, o método eventReceive da classe ButtonDriver da Figura 50 e o método delay da classe Timer da Figura 53. O padrão disposto por (DOUGLASS, 2011) para a solução em código para o método delay

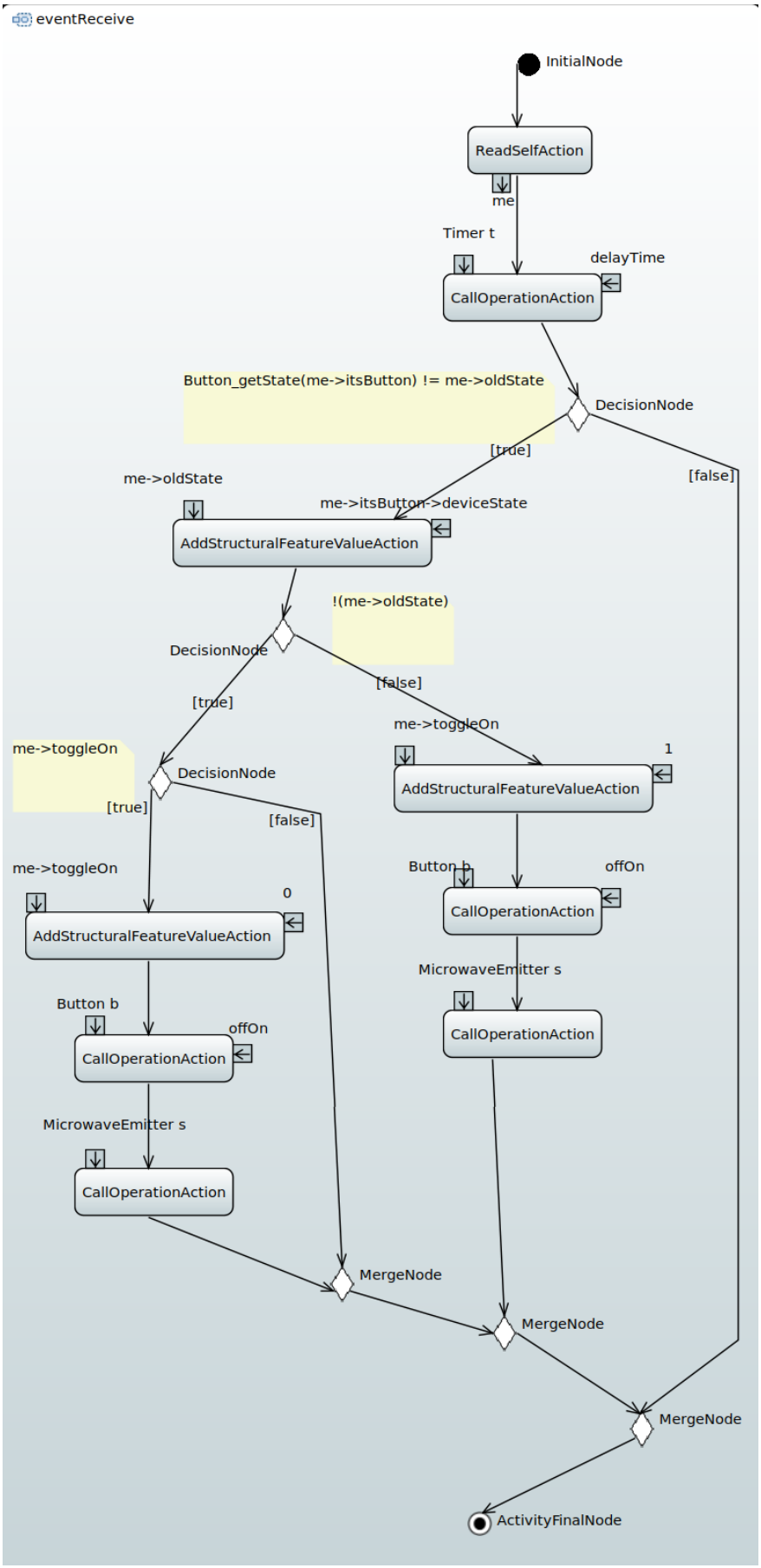
pode ser visualizado na Figura 54 e a solução de (DOUGLASS, 2011) para o método eventReceive é o código disposto na Figura 51. A solução proposta para esses métodos são geradas por meio dos diagramas de atividades correspondentes a cada um, sendo que o código gerado do método delay pode ser observado na Figura 55. E a figura 52 corresponde ao código gerado para o método eventReceive.

Figura 49 – Diagrama de classe ButtonDriver com Eclipse Papyrus Modeling



Fonte: Autor.

Figura 50 – Diagrama de atividade eventReceive da classe ButtonDriver com Eclipse Papyrus Modeling



Fonte: Autor.

Figura 51 – Código idealizado para solução do método eventReceive da classe ButtonDriver

```
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.
           */
       }
}
```

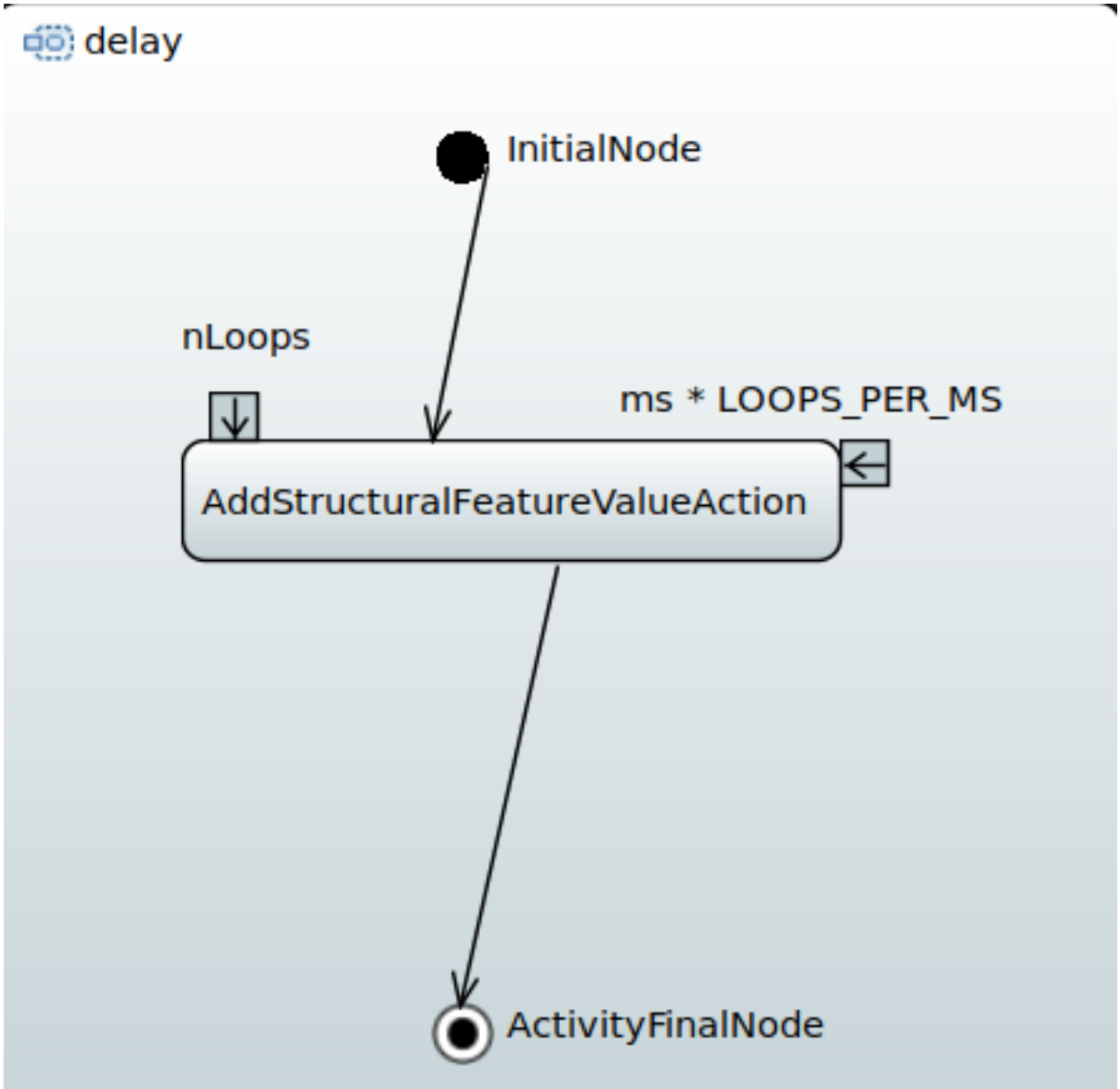
Fonte: Douglass (2011).

Figura 52 – Código gerado para método eventReceive da classe ButtonDriver

```
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();

  }

  }
}
```

Fonte: Autor.

Figura 53 – Diagrama de atividade delay da classe Timer com Eclipse Papyrus Modeling



Fonte: Autor.

Figura 54 – Código idealizado para solução do método delay da classe Timer

```
/* 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-);
    }
```

Fonte: Douglass (2011).

Figura 55 – Código gerado para método delay da classe Timer

```
#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;
}
```

Fonte: Autor.

4.1 LIMITAÇÕES

Após a implementação da geração de código através de modelos para texto observaram-se algumas limitações, como os laços de repetição *for* e *while*. Além disso, ao criar um modelo de classe e posteriormente passa o modelo pela ferramenta Hugo-RT faz com que ele gere automaticamente o método de criação, método de atribuição *set* e retorno dos atributos da classe *get*. A ferramenta Hugo-RT com a adição das implementações que geram código para diagramas de atividade ainda não permite adicionar bibliotecas externas ou efetuar a criação de novos tipos primitivos. Há características de indentação que podem ser melhoradas por uma ferramenta que ajude na visualização da hierarquia. A Figura 56 demostra a aplicação de uma ferramenta de indentação aplicada ao método *writeMotorSpeed*.

Figura 56 – Exemplo de indentação do método writeMotorSpeed

```
void MotorProxy::writeMotorSpeed(direction, int speed) {
    auto mData = this;
    double dPi, dArmLength, dSpeed, dAdjSpeed;
    if (!(this->motorData)) {
        return;
    } else {
        mData.unmarshal();
        mData->direction = direction;
    }
    if (this->rotaryArmLength > 0) {
        dSpeed = speed;
        dArmLength = this->rotaryArmLength;
        dAdjSpeed = dSpeed / 2.0 / 3.14159 / dArmLength * 10.0;
        mData->speed = (int)dAdjSpeed;
    } else {
        mData->speed = speed;
        target.marshal();
        return;
    }
}
```

Fonte: Autor.

As criações de objetos são realizadas por meio da palavra-chave *auto*, que automatiza o processo de especificação do tipo. O destrutor de uma classe que por padrão contém o caractere ~ e o operador & que indica o endereço de memória do objeto não são suportados pela ferramenta Hugo-RT, interpretado na geração pelo carácter _. O gerador não executa a geração de código se o fluxo de controle não começar por um nó inicial e terminar em um nó de atividade final. Alguns modelos apresentaram problemas ao se utilizar o tipo primitivo real, também denominado de

double, impossibilitando a compilação e execução do Hugo-RT. Não é possível realizar a geração de código dos diagramas de atividades fora do comportamento do método.

As equações que caracterizam o arranjo da estrutura do código como a associação de um valor a um objeto necessitam de uma semântica específica, ou seja, a composição da equação deve ser passada diretamente no pino de entrada ou de valor. Outro aspecto relativo à modelagem diz respeito ao comportamento opaco contido na interação *if*, na qual por padrão, a decisão de entrada deve ser um diagrama de atividade, comportamento de função, interação, um comportamento opaco, protocolo de estado de máquina ou um estado de máquina. Sendo assim, a solução que converge para obter uma instrução disposta pelo usuário é o comportamento opaco que permite especificar a linguagem de programação e o trecho de código referente a instrução.

Todo nó *if* deve necessariamente conter dois fluxos de controle mesmo que um dos fluxos vá diretamente ao *MergeNode*. A guarda que representa a parte verdadeira deve ser inserida na modelagem primeiro, e em seguida deve ser inserido o fluxo de controle correspondente a falsa. Para que o usuário visualize a guarda como valor lógico *true* ou *false* é preciso especificar selecionando o fluxo de controle e criando um objeto.

4.2 DISCUSSÕES

Certas características contribuem para uma modelagem mais fluida e viabilizam 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.

O processo de auto geração de código contido na ferramenta Hugo-RT proporciona ao usuário a geração de código dos métodos essenciais para uma classe. Esses métodos que retornam e alteram os valores dos atributos são dispostos de modo que o usuário não precise modelar diretamente, proporcionando economia de tempo. Como contraponto é possível criar um método, mas ele contém um número no final que indica a criação de um novo método, por exemplo, *set0* ou *get1*. Portanto, o indivíduo que realizar a modelagem pode criar um método diferente do usual para retorno e alteração dos atributos.

Em algumas ações é possível especificar uma maior quantidade de informações, e isto possibilita gerar equações em forma de código sem explicitar

diretamente nos pinos de valor ou pinos de entrada. Entretanto, como nem todas as ações têm as mesmas características, optou-se por padronizar as instruções nos pinos de entrada e de valor. Por exemplo, a ação *StructuralFeatureValueAction* possibilita detalhar o tipo do objeto de entrada conforme um atributo da classe, mas não permite determinar um tipo primitivo como a ação *ActivityParameterNode* permite. Além disso, a ação *ReadSelfAction* não permite especificar um tipo.

Claro que a impossibilidade de fazer certas especificações em uma ação corresponde ao que a ação traz para o arranjo da modelagem. Portanto, para padronizar optou-se por definir as instruções em código nos pinos de entrada, saída e valor. Outro ponto a ser levantado é uma falha pontual na demostração de comentários por meio da ferramenta Eclipse Papyrus Modeling, a mesma demostra um bloco de comentário com fundo branco mesmo quando há presença de texto. Nesses casos é necessário realizar um duplo clique no comentário para que esse erro seja solucionado.

4.3 CONSIDERAÇÕES FINAIS

Através das especificações dispostas pela UML, por meio da ferramenta Eclipse Papyrus Modeling é possível efetuar a modelagem do diagrama de classe, exemplo na Figura 49. Posteriormente é necessário modelar um diagrama comportamental de atividade para cada método que se deseja gerar código. Durante a modelagem deve-se criar um fluxo de controle entre os nós de modo que o primeiro nó seja o *InitialNode* e o ultimo nó seja *ActivityFinalNode*. Instruções de código de entrada e saída devem ser dispostas nos pinos de entrada, saída ou valor.

O processo de geração de código através de um modelo UML comportamental de atividade pode ser realizado por meio da ferramenta Hugo-RT. A ferramenta consegue identificar e realizar a geração de código para os nós *InitialNode*, *DecisionNode*, *MergeNode* e *ActivityFinalNode*. Além das ações *ReadStructuralFeatureAction*, *AddStructuralFeatureValueAction*, *ActivityParameterNode*, *ClearStructuralFeatureAction*, *CreateObjectAction*, *DestroyObjectAction* e a ação *CallOperationAction*.

5 CONCLUSÕES

O projeto e desenvolvimento de software embarcado realizado por meio da abordagem dirigida a modelos visa capturar os requisitos envolvidos no projeto e projetar sua arquitetura, evitando ambiguidades. Juntando a abordagem dirigida a modelos com uma ferramenta de geração de código consegue-se produzir código legível e útil para um software. Visto que o ato de modelagem evita o retrabalho proveniente da falha de comunicação no projeto, mesmo que haja tal falha, a alteração na modelagem resulta em uma mudança no código gerado final.

O presente trabalho utilizou a linguagem de modelagem unificada (UML) contida na abordagem dirigida por modelos, para modelar diagramas comportamentais de atividade, além de diagramas estruturais de classe. Teve-se como objetivo a aplicação em desenvolvimento de software para sistemas embarcados, aplicando a modelagem em padrão de *proxy* de hardware, padrão de Adaptador de *Hardware* e padrão de *Debouncing*, gerando assim fragmentos de código de grande parte dos exemplos utilizados.

Foram suportados os seguintes nós: *InitialNode*, *DecisionNode*, *MergeNode* e *ActivityFinalNode*. Além disso, foram suportandas as seguintes ações: *ReadStructuralFeatureAction*, *AddStructuralFeatureValueAction*, *ActivityParameterNode*, *ClearStructuralFeatureAction*, *CreateObjectAction*, *DestroyObjectAction* e a ação *CallOperationAction*. Não foram gerados trechos de código completos devido à falta de ações que realizem, por exemplo, os laços de repetição na ferramenta Hugo-RT. Ou seja, o trabalho conseguiu gerar código para as ações e nós programadas no Hugo-RT.

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 de código e aumentar as ações e nós suportados pela ferramenta Hugo-RT. Pode-se criar um algoritmo capaz de identificar se o objeto enviado, por meio de um fluxo de objeto, é do mesmo tipo tanto na saída quanto na

entrada das ações. Por fim pode-se melhorar aspectos referentes a visualização do código produzido.

REFERÊNCIAS

DOUGLASS, B. P. **Design patterns for embedded systems in C**: an embedded software. Oxford: Elsevier, 2011.

FOUNDATION, T. E. **Eclipse Papyrus Modeling environment**. 2022. https://www.eclipse.org/papyrus/. Acessado em 31/05/2022.

GESSENHARTER, D.; RAUSCHER, M. Code generation for uml 2 activity diagrams. In: **Modelling foundations and applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 205–220. Disponível em: https://link.springer.com/chapter/10.1007/978-3-642-21470-7_15#citeas.

GROUP, O. M. **INTRODUCTION TO OMG'S UNIFIED MODELING LANGUAGE**. 2005. https://www.uml.org/what-is-uml.htm. Acessado em 18/07/2022.

GUEDES, G. T. A. **UML 2**: uma abordagem prática. 2. ed. São Paulo: Novatec Editora, 2011.

HERRINGTON, J. **Code generation in action**. Massachusetts: Manning Publications, 2003.

KLEPPE, A.; WARMER, J.; BAST, W. **MDA explained, the model driven architecture**: practice and promise. 5. ed. Massachusetts: Pearson, 2005.

KNAPP, A. **Hugo/RT**. 2022. https://www.uni-augsburg.de/en/fakultaet/fai/informatik/prof/swtsse/hugo-rt/. Acessado em 31/05/2022.

LIMA, A. d. S. **UML 2.5 : do requisito à solução.** Érica, 2011. ISBN 9788536508320. Disponível em: https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=cat07205a&AN=uls.336689&lang=pt-br&site=eds-live&scope=site&authtype=ip,uid&group=main&profile=eds.

MERZ, S.; KNAPP, A. **Model Checking**: Uml statecharts and collaborations. 2001. https://members.loria.fr/SMerz/talks/hugo-turku.pdf. Acessado em 21/06/2022.

OBJECT MANAGEMENT GROUP. **Unified modeling language**: a specification defining a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems. Massachusetts, 2017. Disponível em: https://www.omg.org/spec/UML/2.5.1/PDF. Acesso em: 25 ago. 2021.

PROJECT, E. M. **Eclipse IDE**. 2022. https://www.eclipse.org/downloads/packages/. Acessado em 12/05/2022.

ROSEN, M. **Introduction to Model Driven Architecture**. 2003. https://www.omg.org/news/meetings/workshops/MDA_2003-2_Manual/Tutorial_1_Rosen.pdf. Acessado em 21/06/2022.

RUMPE, B. **Agile modeling with UML**: code generation, testing, refactoring. Aachen: Springer International Publishing, 2017.

SCHäFER, T.; KNAPP, A.; MERZ, S. Model checking uml state machines and collaborations. **Electronic Notes in Theoretical Computer Science**, v. 55, n. 3, p. 357–369, 2001. ISSN 1571-0661. Workshop on Software Model Checking (in connection with CAV '01). Disponível em: https://www.sciencedirect.com/science/article/pii/S1571066104002622.

SEIDL, M. et al. **UML @ classroom**: an introduction to object-oriented modeling. Heidelberg: Springer, 2015.

SILVA, A. M. R. da; VIDEIRA, C. A. E. **UML**: metodologias e ferramentas case. Lisboa: Centro Atlântico, 2001.

SOARES, M.; VRANCKEN, J. A metamodeling approach to transform uml 2.0 sequence diagrams to petri nets. **Proceedings of the IASTED International Conference on Software Engineering, SE 2008**, p. 159–164, 01 2008.

SUNITHA, E.; SAMUEL, P. Object constraint language for code generation from activity models. In: **Information and software technology**. Cochin: Elsevier B.V., 2018. v. 103, p. 92–111. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584916304190. Acesso em: 12 set. 2021.

VÖLTER, M. et al. **Model-Driven Software Development: Technology, Engineering, Management**. Chichester, UK: Wiley, 2013. (Wiley Software Patterns Series). ISBN 9781118725764.

WHITE, E. **Making embedded systems**: design patterns for great software. Cambridge: O'Reilly Media, 2011.

ÜNSALAN, C.; GüRHAN, H. D.; YüCEL, M. E. **Embedded System Design with ARM Cortex-M Microcontrollers**: Applications with c, c++ and micropython. 1. ed. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer, Cham, 2022.

APÊNDICE A - CÓDIGO UFLOW.JAVA

```
1  public String doTranslation(ActivityNode node) {
2      StringBuilder resultBuilder = new StringBuilder();
3
4      if (node instanceof InitialNode) {
5          resultBuilder.append(new UInitialNode().toStringRoutine());
6      } else if (node instanceof ActivityFinalNode) {
7          resultBuilder.append(new UActivityFinalNode().
               toStringRoutine());
8      } else if (node instanceof ReadSelfAction) {
9          resultBuilder.append(new UReadSelfAction().toStringRoutine
               (((ReadSelfAction) node)));
10     } else if (node instanceof ClearStructuralFeatureAction) {
11         resultBuilder.append(new UClearStructuralFeatureAction()
12                 .toStringRoutine(((ClearStructuralFeatureAction)
                       node)));
13     } else if (node instanceof AddStructuralFeatureValueAction) {
14         resultBuilder.append(new UAddStructuralFeatureValueAction()
15                 .toStringRoutine(((AddStructuralFeatureValueAction)
                        node)));
16     } else if (node instanceof ReadStructuralFeatureAction) {
17         resultBuilder.append(new UReadStructuralFeatureAction()
18                 .toStringRoutine(((ReadStructuralFeatureAction)
                       node)));
19     } else if (node instanceof CreateObjectAction) {
20         resultBuilder
21                 .append(new UCreateObjectAction().toStringRoutine
                       (((CreateObjectAction) node)));
22     } else if (node instanceof DestroyObjectAction) {
23         resultBuilder.append(new UDestroyObjectAction()
24                 .toStringRoutine(((DestroyObjectAction) node)));
25     } else if (node instanceof DecisionNode) {
26         resultBuilder.append(new UDecisionNode()
27                 .toStringRoutine(((DecisionNode) node)));
28     } else if (node instanceof MergeNode) {
29         resultBuilder.append(new UMergeNode()
30                 .toStringRoutine(((MergeNode) node)));
31     } else if (node instanceof ActivityParameterNode) {
```

```
32          resultBuilder.append(new UActivityParameterNode().
               toStringRoutine(((ActivityParameterNode) node)));
33      }
34      return resultBuilder.toString();
35  }
36
37  public String orderAnyone() {
38      StringBuilder resultBuilderFull = new StringBuilder();
39      Stack<DecisionNode> stackDecisionNode = new Stack<>();
40      ActivityNode next = this.node.stream().filter(aux -> aux
           instanceof InitialNode)
41              .collect(Collectors.toList()).get(0);
42      MergeNode auxMergeNode = null;
43
44      if (this.node.stream().allMatch(aux -> aux instanceof MergeNode
           )) {
45          auxMergeNode = (MergeNode) this.node.stream().filter(aux ->
               aux instanceof MergeNode)
46                  .collect(Collectors.toList()).get(0);
47      }
48
49      while (!(next instanceof ActivityFinalNode)) {
50          if (next instanceof DecisionNode) {
51              stackDecisionNode.push((DecisionNode) next);
52          } else if (next instanceof MergeNode) {
53              auxMergeNode = (MergeNode) next;
54              if (!verifyControlCondition(auxMergeNode)
55                      && !(!(auxMergeNode.getIncomings().get(0).
                           getSource() instanceof DecisionNode)
56                              || !(auxMergeNode.getIncomings().get(1)
                                   .getSource() instanceof DecisionNode
                                   ))) {
57                  if (stackDecisionNode.size() != 0) {
58                      resultBuilderFull.append(doTranslation(next));
59                      next = stackDecisionNode.pop().getOutgoings().
                           get(1).getTarget();
60                  }
61                  enableControlCondition(auxMergeNode);
62                  continue;
63              } else {
64                  resultBuilderFull.append("\n} \n");
```

```
65              }
66          }
67
68          if (verifyControlCondition(auxMergeNode)) {
69              disableControlCondition(auxMergeNode);
70          }
71          if (!(next instanceof MergeNode)) {
72              resultBuilderFull.append(doTranslation(next));
73          }
74
75          next = next.getOutgoings().get(0).getTarget();
76      }
77
78      return resultBuilderFull.toString();
79
80  }
```