Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciências da Computação

Uma abordagem dirigida a modelos para apoiar a reengenharia de sistemas críticos utilizando AADL

Cláudio Nazareth Lopes

JUIZ DE FORA
NOVEMBRO, 2018

Uma abordagem dirigida a modelos para apoiar a reengenharia de sistemas críticos utilizando AADL

Cláudio Nazareth Lopes

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciências da Computação

Bacharelado em Ciências da Computação

Orientador: André Luiz de Oliveira

JUIZ DE FORA

NOVEMBRO, 2018

Uma abordagem dirigida a modelos para apoiar a reengenharia de sistemas críticos utilizando AADL

Cláudio Nazareth Lopes

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO.

Aprovada por:

André Luiz de Oliveira
Doutor

Liamara Scorstegagna (DCC)
Doutora

Rosana Teresinha Vaccare Braga (ICMC USP São Carlos)
Doutora

JUIZ DE FORA
28 DE NOVEMBRO, 2018

Aos meus amigos e irmãos.

Aos pais, pelo apoio e sustento.

Resumo

Sistemas críticos são sistemas em que uma falha pode levar a consequências catastróficas como grandes perdas financeiras e danos físicos. Esses sistemas devem atender a requisitos de disponibilidade, segurança e confiabilidade. Padrões de certificação como a ISO 26262, a SAE ARP 4754A e RTCA DO-178C, demandam que as propriedades de segurança desses sistemas sejam verificadas em diferentes níveis de abstração. Atividades de engenharia de segurança como Hazard Analysis and Risk Assessment, component fault analysis e Fault Tree Analysis devem ser realizadas para identificar potenciais ameaças à segurança do sistema e definir mecanismos para evitar ou minimizar o impacto da ocorrência de tais falhas e para produzir artefatos requeridos para a certificação desses sistemas. Técnicas dirigidas a modelos, como CHESS, OSATE AADL & AADL Error Annex, vêm sendo amplamente utilizadas para apoiar o projeto arquitetural e atividades de engenharia de segurança de sistemas críticos. A realização de atividades de engenharia de segurança a partir do código fonte desses sistemas, implementado em linguagens como C#, é um processo custoso e demorado. Dessa forma, há a necessidade de uma abordagem para apoiar a reengenharia do código fonte de sistemas críticos, por exemplo utilizando técnicas dirigidas a modelos como a AADL, para posteriormente realizar atividades de engenharia de segurança nesses sistemas. Neste trabalho de conclusão de curso é apresentado um conjunto sistemático de passos para apoiar a reengenharia de código fonte de sistemas críticos para a linguagem AADL em uma abordagem. A abordagem proposta foi avaliada em um estudo de caso do domínio aeroespacial. Como resultados deste trabalho, foram identificados os mapeamentos preliminares entre elementos de código fonte e suas representações correspondentes em AADL e foram formalizados os passos necessários para realizar a reengenharia do código fonte para modelos AADL. Os mapeamentos identificados foram preliminarmente validados em um estudo de caso do domínio aeroespacial.

Palavras-chave: AADL, sistemas críticos, reengenharia, engenharia de segurança, certificação, desenvolvimento dirigido a modelos.

Abstract

Critical systems are systems in which a failure can lead to catastrophic consequences such as major financial losses and physical damage. These systems must meet availability, security, and reliability requirements. Certification standards such as ISO 26262, SAE ARP 4754A and RTCA DO-178C require that the security properties of these systems be checked at different levels of abstraction. Security engineering activities such as Hazard Analysis and Risk Assessment, component fault analysis, and Fault Tree Analysis should be performed to identify potential threats to system security and to define mechanisms to avoid or minimize the impact of such failures and to produce artifacts required for systems. Techniques addressed to models such as CHESS, OSATE AADL & AADL Error Annex, have been widely used to support architectural design and security engineering activities of critical systems. Performing security engineering activities from the source code of these systems, implemented in languages such as C#, is a costly and time-consuming process. Thus, there is a need for an approach to support the re-engineering of the source code of critical systems, for example using techniques directed to models such as the AADL, to later perform security engineering activities in those systems. In this course completion project is presented a systematic set of steps to support the source code reengineering of critical systems for the AADL language in one approach. The proposed approach was evaluated in a case study of the aerospace domain. As a result of this work, the preliminary mappings between source code elements and their corresponding representations in AADL were identified and the steps necessary to perform source code reengineering for AADL models were formalized. The identified mappings were preliminarily validated in a case study of the aerospace domain.

Keywords: AADL, critical systems, reengineering, safety engineering, certification, model development.

Agradecimentos

A todos os meus parentes, pelo encorajamento e apoio.

Ao professor André Liuz de Oliveira pela orientação, amizade e principalmente, pela paciência, sem a qual este trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

Ao Bruno José Cesário de Almeida Martins Pelo encorajamento, que sem ele não teria me dedicado a faculdade como o necessário.

"Lembra que o sono é sagrado e alimenta de horizontes o tempo acordado de viver".

Beto Guedes (Amor de Índio)

Conteúdo

Lista de Figuras

Listings

Lista de Abreviações

AADL	Architectural Analysis and Description Language
DCC	Departamento de Ciência da Computação
EPF	Eclipse Process Framework
FMES	Failure Modes and Effects Analysis
FTA	Fault Tree Analysis
HARA	Hazard Analysis and Risk Assessment
ISO	Organização Internacional de Normalização
OSATE	Open Source AADL Tool Environment
RTCA	Comissão Técnica de Rádio para Aeronáutica
RFID	Identificação de Rádio Frequência
UFJF	Universidade Federal de Juiz de Fora
SPEM	Software & Systems Process Engineering Meta-Model

1 Introdução

1.1 Contexto

Sistemas críticos são sistemas computacionais que abrangem desde pequenos dispositivos a sistemas complexos de monitoramento de processos industriais (OLIVEIRA et al., 2011). São sistemas que diferem de sistemas convencionais em virtude de uma falha nesse tipo de sistema poder levar a perdas econômicas significativas, danos físicos, ameaça a vida humana e danos ao meio ambiente (ANDRADE, 2007). Dessa forma, sistemas críticos devem atender aos requisitos de segurança, confiabilidade e disponibilidade. Para atender a esses requisitos, padrões de segurança demandam a realização atividades de engenharia de segurança para verificar as propriedades de segurança de um sistema crítico em diferentes níveis de abstração. Os sistemas críticos possuem algumas propriedades tais como: a confiabilidade, manutenibilidade, disponibilidade, proteção e segurança que devem ser verificadas (OLIVEIRA, 2018) antes da liberação de tais sistemas para operação. Exemplos de sistemas críticos são sistemas aviônicos, nos quais uma falha pode causar danos físicos, ameaçar a vida humana e causar danos ao meio ambiente. Sistemas de frenagem em automóveis também são de natureza crítica, uma vez que o mal funcionamento desse sistema pode levar a danos físicos, ameaça a vida humana.

Em virtude da natureza crítica desses sistemas, padrões de segurança como a ISO 26262 para sistemas automotivos, a SAE ARP 4754A e RTCA DO-178C para aeroespaciais, demandam que as propriedades de segurança desses sistemas sejam verificadas em diferentes níveis de abstração. Assim, atividades de engenharia de segurança como *Hazard Analysis and Risk Assessment* (HARA), *component fault analysis e Fault Tree Analysis* (FTA) devem ser realizadas para identificar potenciais ameaças à segurança do sistema e definir mecanismos para evitar ou minimizar o impacto da ocorrência de tais ameaças na segurança global do sistema, e para produzir artefatos requeridos para a certificação desses sistemas.(OLIVEIRA et al., 2011)

Devido a demanda de verificação de propriedades de segurança em diferentes

níveis de abstração, as atividades relacionadas em nível de requisitos são: HARA (Hazard Analysis e Risk Assessment - Análise de Perigos e Avaliação de Risco) e *Allocation of Safety Requirements* (Alocação de Requisitos de Segurança). HARA constitui um conjunto de atividades preliminares no processo de avaliação de segurança, que pode ser realizada após a especificação de requisitos do sistema e caracterização de seu ambiente de operação. O *Hazard Analysis* destina-se para identificar as potenciais ameaças à segurança do sistema, ou seja condições de falha que levam o sistema a um estado inseguro. Durante o *Risk Assessment*, o risco associado à cada ameaça à segurança do sistema é estimado utilizando parâmetros como severidade e probabilidade de ocorrência. *Allocation of Safety Requirements* é realizado a partir da análise dos resultados da classificação dos riscos associados a cada ameaça fornecido pelo HARA.

Dessa forma, as medidas de redução de risco são alocadas à cada *hazard* do sistema identificado na forma de: requisitos de segurança funcional, ou requisitos de integridade de segurança (Safety Integrity Levels - SILs). Um requisito de segurança funcional consiste em uma funcionalidade que deve ser incluída no sistema, por exemplo redundância, com o objetivo de eliminar ou minimizar os efeitos de uma falha na segurança do sistema.

Já um requisito de integridade de segurança consiste na atribuição de um nível de integridade, com base no sistema de classificação de níveis de integridade definido no padrão de segurança alvo, a uma falha em nível de sistema ou de componente. Tal nível de integridade é definido com base em critérios quantitativos/probabilísticos utilizados para classificar o risco associado a cada *hazard* durante o *risk assessment*.

Padrões de segurança como SAE ARP 4754A e DO-178C definem um conjunto de objetivos de segurança, atividades e artefatos a serem produzidos para atender cada nível de integridade.

Em nível de projeto, a atividade de análise de propagação de falhas em nível de sistema pela arquitetura do sistema, utilizando técnicas como *Fault Tree Analysis*(FTA), deve ser realizada. FTA é uma técnica dedutiva que considera a análise de um evento (*top-event*), tipicamente uma falha em nível de sistema (*hazard*), e deduz-se as suas potenciais causas. *Failure Modes and Effects Analysis*(FMEA) é uma atividade aplicada em nível de componentes. FMEA é uma técnica indutiva no qual a análise inicia-se a partir de

modos de falha de componentes por inferir os efeitos dessas falhas na segurança global do sistema.

Técnicas dirigidas a modelos podem ser utilizadas para apoiar tanto o projeto arquitetural quanto atividades de engenharia de segurança de sistemas críticos. O desenvolvimento dirigido a modelos é um termo utilizado para definir processos de engenharia de software que utilizam modelos como principais artefatos no ciclo de vida de desenvolvimento desses sistemas(PASINI et al., 2008). O processo de desenvolvimento dirigido a modelos possui o enfoque na modelagem do sistema. A principal diferença entre o desenvolvimento dirigido a modelos e o desenvolvimento padrão está no grau de transformação de modelos(FIGUEIREDO, 2011). Ao utilizar o desenvolvimento dirigido a modelos tem-se vantagens de produtividade, portabilidade, interoperabilidade, menor custo e facilidade na evolução do sistema. CHESS e OSATE AADL & Error Annex são exemplos de técnicas dirigidas a modelos. AADL é uma linguagem de modelagem que apoia a especificação da arquitetura de um sistema por meio de uma notação orientada a fluxo de dados (data-flow) com sintaxe e semântica bem definida (FEILER; GLUCH; HUDAK, 2006). AADL apoia a modelagem de componentes de hardware e software e suas interações.

Para a realização das atividades da engenharia da segurança em sistemas que foram diretamente especificados/implementados em código fonte é necessária realização de reengenharia desses sistemas para modelos. Reengenharia de sistemas é o processo de abstrair suas funcionalidades a partir de um sistema existente e são construídos os modelos de análise e o projeto de software sem grandes alterações na funcionalidade do sistema (PALMA; QUINÁIA, 2010). A reengenharia consiste na reorganização e modificação de sistemas de software existentes, de forma parcial ou total, para torná-los mais manuteníveis e extensíveis(SOMMERVILLE, 2001).

1.2 Motivação/Problema

A realização de atividades de engenharia de segurança a partir do código fonte de um sistema crítico, implementado em linguagens como C#, C++ ou JAVA, é um processo custoso e demorado. Portanto, há a necessidade da proposta de uma abordagem para apoiar a reengenharia do código fonte de sistemas críticos, por exemplo utilizando técnicas

dirigidas a modelos como a AADL, para posteriormente realizar atividades de engenharia de segurança nesses sistemas de modo a possibilitar a identificação de potenciais ameaças à segurança de sistemas críticos e a implementação de medidas de mitigação de riscos. Há também na literatura uma carência de abordagens que apoiam a reengenharia de código fonte para modelos.

1.3 Objetivos/Contribuições

Neste trabalho de conclusão de curso é proposta uma abordagem dirigida a modelos para apoiar a reengenharia do código fonte de sistemas críticos, desenvolvidos em linguagem como C++, C# e JAVA para modelos AADL. Desta forma, facilitar o processo de engenharia de segurança de sistemas com o apoio de técnicas dirigidas a modelos.

1.4 Metodologia

Dessa forma, neste trabalho de conclusão de curso foram identificados os mapeamentos preliminares entre elementos de código fonte e suas representações correspondentes em AADL e formalizados os passos necessários para realizar a reengenharia do código fonte para modelos AADL. Os mapeamentos identificados foram preliminarmente validados em um estudo de caso do domínio aeroespacial.

Este trabalho de conclusão de curso possui caráter teórico prático no qual foi realizada uma revisão de literatura para identificar as técnicas existentes para apoiar o processo de reengenharia de sistemas críticos a partir do código fonte. Com base nos achados na literatura, foi elaborada uma abordagem dirigida a modelos para apoiar a reengenharia desses sistemas. A abordagem proposta foi aplicada/validada no processo de reengenharia do sistema critico ArduPilot, um sistema de controle de vôo do domínio aeroespacial.

1.5 Organização da monografia

Este trabalho de conclusão de curso está organizado em cinco capítulos. No Capítulo 2 são apresentados os conceitos necessários ao entendimento do trabalho como: sistemas críticos, padrões de segurança, engenharia de segurança, reengenharia de sistemas críticos, desenvolvimento dirigido a modelos, AADL e modelagem de processo de software. No Capítulo 3 é apresentado a abordagem proposta para apoiar a reengenharia do código fonte para modelos AADL. No Capítulo 4 é apresentada aplicação da abordagem proposta em um estudo de caso do domínio aeroespacial. Finalmente, no Capítulo 5 são apresentadas as conclusões.

2 Revisão Bibliográfica

Nesse capítulo é apresentada uma visão geral dos conceitos como reengenharia de sistemas, sistemas críticos, engenharia de segurança, desenvolvimento dirigido a modelos e uma descrição da linguagem AADL.

2.1 Sistemas Críticos

Os sistemas críticos são sistemas computacionais em que falhas podem causar sérios danos ao sistema, ao usuário ou ao meio ambiente (ANDRADE, 2007). Sistemas críticos são sistemas computacionais que abrangem desde pequenos dispositivos à sistemas complexos de monitoramento de processos industriais (OLIVEIRA et al., 2011). Sistemas Aviônicos, sistemas automotivos e sistemas de dispositivos de marcapasso são exemplos de sistemas críticos. Esses sistemas devem atender a requisitos de segurança, confiabilidade e disponibilidade. Padrões de segurança de diferentes domínios como automotivo e aerospacial demandam ou recomendam a realização de atividades de engenharia de segurança para analisar e verificar as propriedades de segurança do sistema em diferentes níveis de abstração. Os sistemas críticos devem atender às seguintes propriedades:

- Confiabilidade: probabilidade de um sistemas operar corretamente desde sua inicialização;
- Manutenibilidade: probabilidade de um sistema operar corretamente uma certa unidade de tempo após a ocorrência de uma falha ou erro;
- Disponibilidade: probabilidade de um sistema operar corretamente sobre um determinado conjunto de condições em um certo tempo;
- Proteção: a operação do sistema não causará nenhum prejuízo ao ambiente em que está operando;
- Segurança: proteção do sistema contra potenciais ameaças do ambiente no qual o

sistema está operando.

2.2 Padrões de Segurança

Em virtude da natureza crítica de sistemas embarcados, padrões de segurança como a ISO 26262 para sistemas automotivos e SAE ARP 4754A e DO-178C para sistemas aviônicos demandam a realização de atividades de engenharia de segurança para analisar e verificar as propriedades de segurança como pré-requisitos para obter a certificação desses sistemas e liberação para operação.

O padrão ISO 26262 fornece diretrizes para o desenvolvimento de sistemas automotivos confiáveis. Os principais objetivos da ISO 26262 são: fornecer um ciclo de vida de engenharia de segurança e apoiar a adaptação das atividades durante esse ciclo de vida, fornecer uma abordagem de análise de riscos para determinar as classes de riscos ou níveis de integridade, possibilitando a utilização de tais classes de risco para especificar requisitos de segurança para eliminar ou minimizar os efeitos da ocorrência de falhas sobre a segurança global do sistema (SINHA, 2011).

O RTCA DO-178C é o padrão de segurança para sistemas aviônicos. O padrão DO-178C define um conjunto de atividades e artefatos de desenvolvimento que devem ser produzidos para atingir a certificação. O DO-178C prescreve um conjunto de atividades específicas que devem ser realizadas para satisfazer objetivos de segurança específicos de acordo com o nível de integridade alvo (HOLLOWAY, 2012). O padrão SAE ARP 4754A define um conjunto de atividades de engenharia de segurança que devem ser realizadas e artefatos a serem produzidos para se obter a certificação de um sistema crítico de acordo com o nível de integridade alvo.

2.3 Engenharia de Segurança

As propriedades de segurança devem ser atendidas em diferentes níveis de abstração: Em nível de requisitos deve-se identificar as potenciais ameaças à segurança do sistema chamadas *hazards* e classificar os riscos associados a cada ameaça. Em nível de projeto, é necessário analisar como as falhas em nível de sistema se propagam pela arquitetura

utilizando técnicas como análise de árvore de falhas. Em nível de componentes, deve-se analisar como cada componente contribui diretamente ou indiretamente para a ocorrência de falhas em nível de sistema.

Hazard identification(identificação de perigos) é uma atividade preliminar no processo de análise de segurança, que pode ser realizada logo após a especificação de requisitos do sistema e da caracterização de seu ambiente de operação. A atividade de identificação de perigos tem o objetivo de identificar as condições de falhas que podem conduzir o sistema a um estado inseguro. *Hazard identification* pode ser executado junto com a evolução do sistema (OLIVEIRA et al., 2011).

Durante a atividade de *risk assessment* (avaliação de riscos), os fatores de risco associados à cada *hazard* são determinados em termos de atributos como severidade e probabilidade de ocorrência. No processo de classificação de riscos, é necessário estabelecer critérios de tolerância a falhas e demonstrar que os riscos associados a cada hazard são "*As Low as Reasonably Practicable*"(ALARP). Para um risco ser considerado ALARP o custo de medidas de redução de riscos deve ser desproporcional ao benefício obtido a partir dessas medidas. Os riscos podem ser classificados como:

- Intolerável: O risco deve ser reduzido independente do custo;
- Tolerável: Satisfaz o ALARP;
- Amplamente aceitável: O risco é aceitável desde que o sistema aborde boas práticas de programação.

Allocation of Safety Requirements (Alocação de Requisitos de Segurança) é realizada a partir dos resultados das atividades de *Hazard Analysis* e *Risk Assessment*. Dessa forma, medidas de redução de riscos são alocadas a cada *hazard* identificado na forma de requisitos de segurança funcionais ou Safety Integrity Levels (SILs).

A IEC 61508 estabelece requisitos de integridade de segurança em termos de duas categorias de taxas de falhas: probabilidade de falha no desempenho de funções de alta demanda/contínuas e probabilidade de falha na execução de funções sob demanda. Dessa forma, cada um dos Níveis de Integridade de Segurança (SILs), mostrados na Tabela 2.1,

está associado a faixas de taxas de falhas em ambas as categorias. O SIL 4 é o mais rigoroso e o SIL 1 o menos rigoroso.

Safety Integrity Level	Continuous Probability of dangerous failure per Hour	On Demand Probability of failure to perform the Function
4	$10^{-9} < P \leq 10^{-8}$	$10^{-4} < P \leq 10^{-5}$
3	$10^{-8} < P \leq 10^{-7}$	$10^{-3} < P \leq 10^{-4}$
2	$10^{-7} < P \leq 10^{-6}$	$10^{-2} < P \leq 10^{-3}$
1	$10^{-6} < P \leq 10^{-5}$	$10^{-1} < P \leq 10^{-2}$

Figura 2.1: Critérios probabilísticos e níveis de integridade de segurança na IEC 61508

Não há um consenso para a avaliação de riscos de um sistema crítico, pois os padrões podem mudar de acordo com o domínio e região. Por esse motivo, diferentes padrões de segurança de sistemas críticos devem ser seguidos de acordo com o domínio de aplicação alvo, como a ISO 26262 para sistemas do domínio automotivo e RTCA DO-178C para sistemas aviônicos.

2.4 Reengenharia de Sistemas

A Reengenharia de sistemas é o processo de abstrair suas funcionalidades a partir de um sistema existente e construir modelos de análise e de projeto sem grandes alterações na funcionalidade do sistema (PALMA; QUINÁIA, 2010) que passa por reengenharia. Dessa forma, a reengenharia consiste na reorganização e modificação de sistemas de software existentes, de forma parcial ou total, para torná-los mais fácil de se aplicar a manutenção (SOMMERVILLE, 2001). A reengenharia é definida como um subconjunto da Engenharia de Software, composto por técnicas de Engenharia Reversa e Engenharia Progressiva.

2.5 Desenvolvimento Dirigido a Modelos

O desenvolvimento dirigido a modelos fornece suporte para a análise e transformação automática de modelos, por definir as abstrações associadas a um dado modelo em uma linguagem com sintaxe e semântica bem definida chamada metamodelo (SELIC, 2003). O Processo de Desenvolvimento Dirigido a Modelos possui o enfoque na modelagem do

sistema. A principal diferença entre o Desenvolvimento Dirigido a Modelos e o desenvolvimento convencional está no grau de transformação de modelos (FIGUEIREDO, 2011). Ao utilizar o desenvolvimento dirigido a modelos tem-se as seguintes vantagens: produtividade, portabilidade, interoperabilidade, menor custo e maior facilidade na evolução do sistema. CHESS e OSATE AADL & AADL Error Annex são exemplos de técnicas dirigidas a modelos que apoiam a engenharia de sistemas críticos.

2.6 Architectural & Analysis Description Language (AADL)

AADL é uma linguagem de modelagem que apóia a modelagem arquitetural do sistema baseada em uma notação orientada a fluxo de dados (data-flow) com sintaxe e semântica bem definida. AADL apoia a especificação de componentes de hardware e de software e de suas interações.

AADL foi inspirada na linguagem MetaH, desenvolvida pela Honeywell uma multinacional estadunidense que produz sistemas aeroespaciais. Inicialmente, AADL era chamada de *Avionics Architecture Description Language*(Linguagem de Descrição de Arquitetura de Aviônicos) pois foi originalmente projetada para especificar sistemas aviônicos. Após o interesse de outro grupos de pesquisas e empresas, o seu nome foi modificado para *Architectural Analysis and Design Language*(Linguagem de Análise e Projeto Arquitetural). Atualmente, AADL apóia a especificação de arquiteturas de sistemas críticos e sistemas de tempo real.

Os Modelos especificados em AADL não se preocupam com a implementação das funções, apenas especifica as conexões, associações entre interfaces e propriedades não funcionais de sistemas. AADL fornece três categorias de componentes: componentes de *software*, componentes de *hardware* e componentes do tipo *sistema*. Componentes do tipo *sistema* são *containeres* para componentes de hardware e de software. Componentes de hardware incluem: processadores (*processor*), conectores (*buses*), memórias (*memory*) e dispositivos (*device*). Os componentes de software abrangem: threads, *thread group* (grupos de threads), *process*(processo), *data*(dados) e *subprogram*(subprogramas)(BRUN; DE-

LATOUR; TRINQUET, 2008). Esses componentes podem ser utilizados para representar uma aplicação de software ou hardware.

Em AADL, um componente possui uma definição e pode conter múltiplas implementações. A definição ou declaração de um componente inclui a especificação dos recursos do componente como portas de entrada e de saída. A implementação de um componente inclui a especificação das conexões entre portas de entrada e de saída e conexões entre subcomponentes especificados na definição do componente. Os recursos incluem dados, portas e sub-rotinas. A linguagem AADL permite a adição de propriedades a componentes, conexões e *features*. As propriedades são atributos que determinam restrições ou características que se aplicam a elementos do modelo arquitetural. Por exemplo, a propriedade "clock frequency"pode ser associada a um componente do tipo processador, a propriedade tempo de execução pode ser associada a uma *thread* e a propriedade largura de banda pode ser associada a um componente do tipo *bus*. O conceito de herança da orientação a objetos é aplicável à AADL de modo semelhante às linguagens de programação orientadas a objetos. (MA et al., 2008)

2.6.1 Componentes de sistema

Componentes do tipo *system* (sistema) representam a integração de componentes de hardware e de software. Na Figura 2.2 é mostrada a representação gráfica da especificação de um componente do tipo sistema em AADL. Na listagem 2.1 é mostrada a representação textual da implementação de um componente do tipo sistema em AADL.

Figura 2.2: Representação gráfica do sistema

Listing 2.1: Representação textual do sistema

```
1 System sistema
2 end sistema;
3
4 System implementation sistema.impl
5 end sistema.impl
```

2.6.2 Componentes de Software

Os componentes de software podem ser dos seguintes tipos: *process*, thread, *thread group*, *data* e *subprogram*.

Componentes do Tipo Processo

Processos são espaços de memória virtual, onde estão contidas estruturas de dados e códigos executáveis. Na declaração de um processo são permitidos três tipos de subcomponentes: thread, *thread group* e *data*. Também é possível especificar a um processo as seguintes propriedades: proteção de memória, informações do código executável e protocolos de escalonabilidade. Na Figura 2.3 é mostrada a representação gráfica de um componente do tipo processo em AADL. Na listagem 2.2 é mostrada a representação textual da implementação de um componente do tipo processo em AADL.

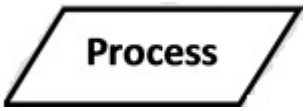

Figura 2.3: Representação gráfica do *process*

Listing 2.2: Representação textual do *process*

```
1  Process processo
2      features
3          identVariavel: tipoDaPorta nomeDaPorta;
4          timeout => 5ms;
5      properties
6          --propriedades do processo
7  end processo;
8
9  Process implementation processo.impl
10     subcomponents
11         identVatiavel: tipoDoComponente nomeDoComponete.impl;
12     connections
13        tipoComponente.nomePorta -> ipoComponente.nomePorta;
14 end processo.impl
```

Componentes do Tipo Thread

Thread é um componente de execução sequencial de código fonte. Uma Thread pode conter apenas componentes do tipo *data*. Uma Thread é declarada juntamente com as suas

propriedades de execução, por exemplo, prioridade e tempo de execução. O componente *process* pode conter uma ou mais *Threads*. Na Figura 2.4 é mostrada a representação gráfica de um componente *Thread* em AADL. A representação textual da implementação de um componente thread é mostrada na listagem 2.3.

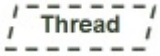

Figura 2.4: Representação gráfica da thread

Listing 2.3: Representação textual da thread

```
1  Thread tarefa1
2      features
3          identVariavel: in out data port tipoPorta;
4          timeout => 5ms;
5      properties
6          --propriedades da thread
7  end tarefa1;
8
9  Thread implementation tarefa.impl
10     properties
11         Priority => 1;
12         Compute_execution_time => 0ms .. 3ms;
13     connections
14        C0: parameter identVariavel -> tipoComponente.
             nomePortaDestino;
15 end tarefa.impl
```

Componentes do Tipo Data

Componentes do tipo *data* são usados na declaração de tipos de dados do sistema. Na Figura 2.5 é mostrada a representação gráfica da especificação de um componente do tipo *data* em AADL. A representação textual da implementação de um componente do tipo *data* é mostrada na listagem 2.4.

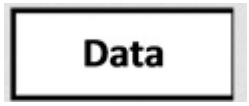

Figura 2.5: Representação gráfica do *data*

Listing 2.4: Representação textual do *data*

```
1  data protected_data
```

```
    properties
        Concurrency_Control_Protocol => Protected_Access;
end protected_data;
```

Componentes do Tipo Subprograma

Um *subprograma* representa trechos de código de execução sequencial. As chamadas a um subprograma podem ser feitas por um componente do tipo *thread* ou por outros *subprogramas*. Um subprograma não permite a declaração de subcomponentes. Um subprograma pode conter a declaração de propriedades relacionadas a requisitos e mapeamentos da memória e chamadas de subprogramas. Na Figura 2.6 é mostrada a representação gráfica de um subprograma em AADL. Na listagem 2.5 é mostrada a representação textual da implementação de um componente *subprogram* em AADL.

Figura 2.6: Representação gráfica do *subprogram*

Listing 2.5: Representação textual do *subprogram*

```
subprogram subprograma
    features
        x: out parameter number;
        y: out parameter numver;
        z: in parameter number;
end subprograma;

subprogram implementation subprograma.imp
    subcomponents
          local_K: data number;
     calls
          Mycall: {
               S1: subprogram subprograma1;

          };
     connections
          C0: parameter x -> S1.x;
          C1: parameter y -> S1.y;
          C2: parameter S1.z -> z;
end subprograma.imp;
```

2.6.3 Componentes de Hardware

Os componentes de software devem ser mapeados para componentes de hardware . Componentes de hardware definem onde o código fonte é armazenado e executado.

Componentes do Tipo Processor

Um componente do tipo processador (*processor*) é responsável pela execução do código fonte. Esse componente pode conter componentes do tipo memory, e pode conter a especificação de propriedades relacionadas à protocolos de escalonamento e descrição de hardware. Na Figura 2.7 é mostrada a representação gráfica da especificação de um componente do tipo *processor* em AADL. Na listagem 2.6 é mostrada a representação textual AADL da implementação de um componente do tipo *processor*.

Figura 2.7: Representação gráfica do *processor*

Listing 2.6: Representação textual do *processor*

```
1 processor processador
2     properties
3         Scheduling_Protocol =>
              POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL;
4 end processador;
```

Componentes do Tipo Memory

Memory representa um componente de armazenamento de dados e de código fonte. Componentes do tipo *memory* podem conter apenas outros componentes *memory* como subcomponentes, e suas propriedades estão relacionadas a: tamanho da palavra e tipos de acesso. Na Figura 2.8 é mostrada a representação gráfica da especificação de um componente do tipo *memory*. Na listagem 2.7 é mostrada a representação textual AADL da implementação de um componente do tipo *memory*.

Listing 2.7: Representação textual da *memory*

```
1 memory memoria
```

Figura 2.8: Representação gráfica da *memory*

```
2     properties
3          Memory_Type => (Data_Memory, Code_Memory);
4 end memoria;
```

Componentes do Tipo Device

Os componentes do tipo *device* são responsáveis pela comunicação com dispositivos externos e podem ser de três tipos: um componente físico, parte de um sistema ou uma unidade controlável. As propriedades de um device estão relacionadas às restrições de software e de hardware. Na Figura 2.9 é mostrada a representação gráfica da especificação de um componente do tipo *device* em AADL. A representação textual AADL da implementação desse componente é mostrada na listagem 2.8.

Figura 2.9: Representação gráfica do *device*

Listing 2.8: Representação textual do *device*

```
1 device dispositivo
2     features
3         busin: requires bus access ;
4         displaydata: in event data port ;
5 end dispositivo;
```

Componentes do Tipo Bus

Um *bus* representa o barramento de comunicação de um *processor*, de um *memory* e/ou de um *device*. As propriedades de um componente do tipo *bus* estão relacionadas às características de comunicação, como protocolo de acesso e tempo de transmissão. Um bus não permite a declaração de subcomponentes e o único componente que pode conter sua declaração é um componente do tipo *system*. Na Figura 2.10 é mostrada a repre-

sentação gráfica da especificação desse componente em AADL. A representação textual da implementação de um componente do tipo *bus* é mostrada na Listagem 2.9.

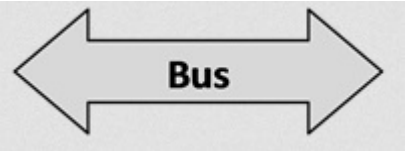

Figura 2.10: Representação gráfica do *bus*

Listing 2.9: Representação textual do *bus*

```
1 bus ethernet
2 end ethernet;
```

Na Figura 2.11 ilustra um diagrama exemplo dos componentes e suas interações, onde estão representados os componentes *package* Pacote exemplo, *process* ClasseExemplo e sua implementação com a *features* x e *subcomponents* funcao1 do tipo *thread* e funcao2 do tipo *subprogram*.

O componente *thread* funcao1 e sua implementação com *features* *b* e *i* do tipo *data port*, *subcomponents* *f* do tipo *data* e uma chamada *MyCall*.

O componente *subprogram* funcao2 e sua implementação com *features* *I* do tipo *in parameter*.

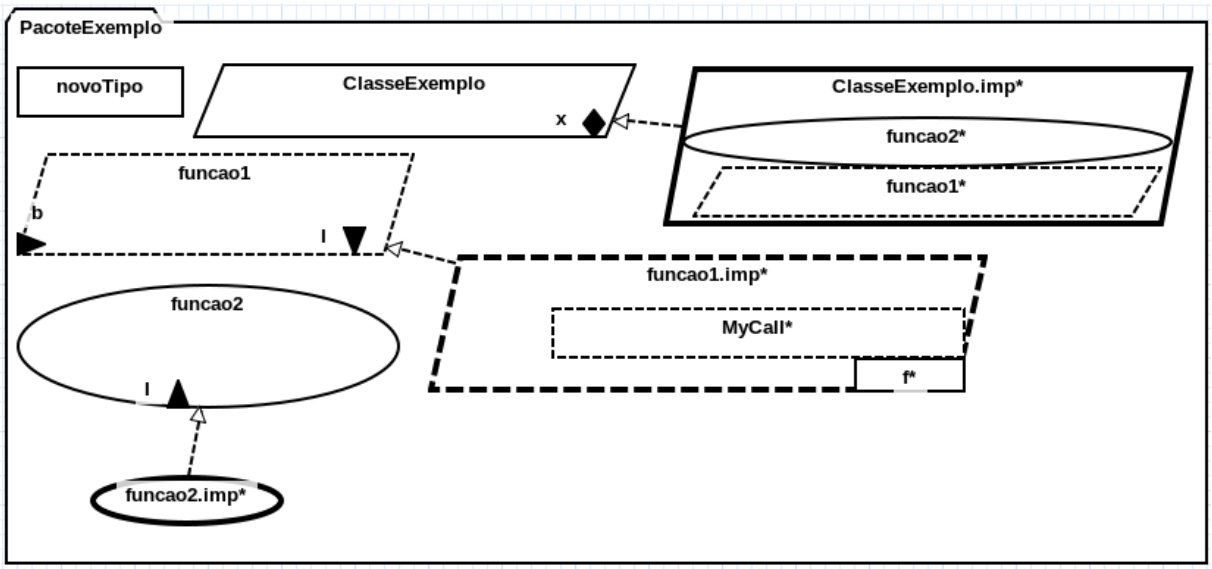

Figura 2.11: Diagrama AADL

2.7 Modelagem de Processo de Software

Nesta seção é apresentada a especificação SPEM 2.0 e a ferramenta EPF Composer utilizadas na modelagem de processo de reengenharia proposto neste trabalho de conclusão de curso.

2.7.1 SPEM

SPEM 2.0 define uma notação para a modelagem de processos de software baseada na notação do Diagrama de Atividades da UML (Unified Modeling Language). A especificação SPEM 2.0 possibilita a especificação semi-formal de uma gama de métodos e processos de desenvolvimento, facilitando o entendimento de processos de software, a adaptação de processos de software e a gerência de processos de software (OBJECT MANAGEMENT GROUP, 2008).

A especificação de um modelo de processo em SPEM 2.0 pode conter os seguintes elementos mostrados na Figura 2.11:

- Processo: representa uma instância de um processo padrão para um projeto específico;
- Fase: representa um período significativo de um processo instanciado;
- Iteração: representa um conjunto de atividades e tarefas que são repetidas em uma fase;
- Atividade: representa um trabalho a ser realizado durante um processo instanciado e que pode ser decomposto em tarefas tasks;
- Tarefa: representa uma unidade de trabalho que não pode ser decomposta.

2.7.2 EPF Composer

EPF Composer(*Eclipse Process Framework Composer*) é uma ferramenta baseada na plataforma Eclipse que apóia a especificação, documentação, adaptação e publicação de modelos de processos de software em SPEM 2.0 (ECLIPSE FOUNDATION, 2018). O EPF

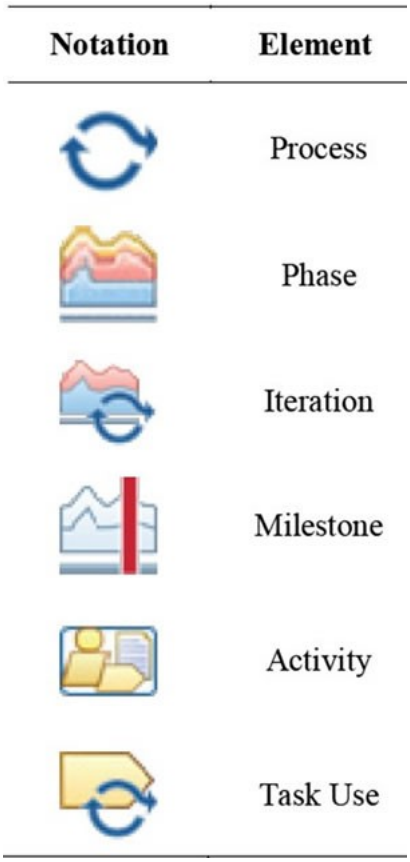

Figura 2.12: Notação SPEM 2.0 para modelagem de processo

Composer fornece uma estrutura extensível para engenharia de processos de software, criação de métodos e processos, gerenciamento de bibliotecas, configuração e publicação de um processo. O EPF oferece suporte ao desenvolvimento iterativo, ágil e incremental. Uma particularidade do EPF Composer é a categorização dos elementos da especificação SPEM 2.0 em: *Method Content* e *Process*.

O *Method Content* descreve o que deve ser produzido (*WorkProduct*), as competências necessárias exigidas (*roles*), e as instruções passo-a-passo (*steps*), explicando como os objetivos serão atingidos. Um *Process* descreve o ciclo de vida do desenvolvimento, relacionando os elementos dos *Method Contents* em sequências semi-ordenadas que são personalizadas para determinados tipos de projetos.

2.8 Trabalhos Relacionados

Há na literatura uma carência de abordagens que apoiam a reengenharia de código fonte para AADL. Porem, existem dois trabalho que realiza o mapeamento da linguagem AADL

para a linguagem C. Este trabalho de conclusão de curso realiza o mapeamento inverso, convertendo um código em C# para AADL.

O trabalho do (MA et al., 2008) projeta um gerador de código automático baseado no modelo AADL, apresenta a regra de mapeamento do elemento do modelo AADL para o da linguagem C e conduz o teste de validação com a combinação do sistema DELTA OS, um sistema operacional integrado em tempo real com sistema intelectual independente direitos de propriedade, e LAMDA PRO, uma plataforma de desenvolvimento de integração, prova a validade de gerar o programa C automaticamente a partir do modelo AADL.

O (GUI et al., 2008). assim como o (MA et al., 2008) apresenta as regras de mapeamento entre AADL e a linguagem C, projetaram um gerador de código C automático chamado UCaG e pode ser integrada no OSATE, que é de código aberto e suporta uma representação interna baseada em Modelos AADL, usamos um caso desenvolvido pela ferramenta Lambda que é o ambiente de desenvolvimento e simulação do Delta OS para mostrar a correção do UCaG.

3 Uma Proposta de Abordagem de Apoio à Reengenharia de Sistemas Críticos

Neste capítulo são apresentadas as regras de mapeamento entre elementos de código fonte e sua respectiva representação em AADL e uma proposta de abordagem de apoio à reengenharia de sistemas críticos.

3.1 Mapeamento Entre Código Fonte C# para Representações AADL

Nesta seção são apresentadas as regras de mapeamento entre elementos do código fonte na linguagem C# para as respectivas representações em AADL. Os mapeamentos de representações em código fonte para AADL foram extraídos a partir da análise de múltiplos artefatos de códigos fonte do ArduPilot, um sistema de controle de vôo implementado na linguagem C#.

Para o mapeamento entre código fonte para AADL, nem tudo que está no código pode ser representado em AADL, o corpo do dos métodos não são mapeados para AADL. São mapeados apenas funções, variáveis, classes, pacotes, conexões entre classes ou entre funções e alguns conceitos de orientação a objetos como abstração e generalização/especialização.

3.1.1 Mapeamento de Pacotes

Um pacote na linguagem C# é chamado de namespace e pode ser mapeado para um elemento AADL do tipo *Package*. Nas Listagens 3.1 e 3.2 são mostradas respectivamente a representação de um pacote na linguagem C# e sua representação em AADL.

Listing 3.1: Namespace em C#

```
1 namespace CodigoExemplo.PacoteExemplo{
2
3 }
```

Listing 3.2: Package em AADL

```
1 package PacoteExemplo
2 public
3
4 end PacoteExemplo;
```

3.1.2 Mapeamento de Classes

A especificação de uma *classe* na linguagem C# é mapeada para a especificação e a implementação de um componente de software AADL do tipo *Process*. Nas Listagens 3.3 e 3.4 são mostradas respectivamente a representação de uma classe na linguagem C# e sua respectiva representação em AADL, que inclui a especificação do processo (linhas 3 e 4) e a implementação do processo nas linhas 6 e 7.

Listing 3.3: Classe em C#

```
1 public class ClassExemplo {
2   //Conteudo da classe
3 }
```

Listing 3.4: *Process* em AADL

```
1 package PacoteExemplo
2 public
3     process ClassExemplo
4     end ClassExemplo;
5
6     process implementation ClassExemplo.imp
7     end ClassExemplo.imp;
8 end PacoteExemplo;
```

3.1.3 Mapeamento de Declarações de Importações(*imports*) dentro de Classes

Da mesma forma que a linguagem C# e outras linguagens de programação como Java permitem a especificação de importação de classes e bibliotecas, no caso do C# a importação é por meio da sintaxe *using* (Listagem 3.5), AADL possibilita a especificação de declarações *import* dentro de AADL *Packages* utilizando a palavra reservada *with* no cabeçalho da especificação de um AADL Package, como mostrado na listagem 3.6. A utilização da biblioteca importada é feita por quatro pontos(::).

Listing 3.5: *Import* em C#

```
1 using classe1;
```

Listing 3.6: *With* em AADL

```
1 with classe1;
```

3.1.4 Mapeamento de Variáveis/Atributos de uma Classe

Os tipos de dados primitivos **int**, **float** e **Boolean** são suportados por AADL, sendo necessário importar tais tipos da biblioteca Base Types de AADL. Da mesma forma que linguagens como C# e Java, AADL permite a declaração de novos tipos de dados, ou seja, tipos por referência, utilizando o componente *data* da linguagem AADL. A declaração de novos tipos de dados em AADL é feita dentro de um AADL *Package* e não em um componente do tipo *Process*. Nas Listagens 3.7 e 3.8 são mostradas respectivamente a representação da declaração de três variáveis na classe ClasseExemplo na linguagem C# e seu respectivo mapeamento em AADL, que inclui a especificação do processo ClasseExemplo (linhas 3 à 12) com a declaração das features x, variável e tipoDado do tipo in out data port. O tipo de dado por refrencia NovoTipo dentro da ClasseExemplo em C# é mapeado para um componente do tipo *data* declarado dentro do AADL Package PacoteExemplo (linhas 5 e 6 da listagem 3.8). Resumindo, variáveis globais de uma classe em C# são mapeadas para portas declaradas dentro do campo features de um componente do tipo process.

Listing 3.7: Variáveis/Atributos em C#

```
1 public class ClassExemplo {
2     Class2 variavel;
3     int x;
4     NovoTipo tipoDado;
5 }
```

Listing 3.8: Variáveis/Atributos em AADL

```
1 package PacoteExemplo
2 public
3     with Base_Types, Class2;
4
5     data NovoTipo
```

```
6       end NovoTipo;
7
8       process ClasseExemplo
9          features
10          x: in out data port Base_Types::Integer;
11          variavel: in out data port Class2::variavel;
12          tipoDado: in out data port NovoTipo;
13      end ClasseExemplo;
14
15  end PacoteExemplo;
```

3.1.5 Mapeamento de Funções

A declaração de uma função em linguagens de programação orientadas a objetos com C# e Java, como mostrado na listagem 3.9, pode ser mapeada para a declaração de um componente AADL do tipo *Thread* ou *subprogram* como subcomponente de um *Process implementation* (linhas 7 e 8 da Listagem 3.10). Primeiramente, a *função1* declarada na classe ClasseExemplo da listagem 3.9 é mapeada para a especificação da *thread* funcao1 (linhas 11 e 12 da listagem 3.10) e sua respectiva implementação (linhas 14 e 15 da listagem 3.10). Posteriormente, uma variável do tipo thread funcao1 é declarada dentro da implementação do componente do tipo *Process* ClasseExemplo (linhas 7 e 8 da listagem 3.10).

Listing 3.9: Função simples em C#

```
1  public class ClassExemplo {
2    public void funcao1(){
3
4    }
5  }
```

Listing 3.10: thread simples em AADL

```
1  package PacoteExemplo
2  public
3      process ClassExemplo
4      end ClassExemplo;
5
6      process implementation ClassExemplo.imp
7          subcomponents
8              F1: thread funcao1;
9      end ClassExemplo.imp;
10
11     thread funcao1
```

```
12      end funcao1;
13
14      thread implementation funcao1.imp
15      end funcao1.imp;
16  end PacoteExemplo;
```

A separação de funções públicas e privadas presentes em linguagens orientadas a objetos pode ser diretamente mapeada para AADL. Entretanto, em AADL, os componentes declarados como públicos não podem interagir com componentes privados, mas componentes privados podem interagir com componentes declarados como públicos por meio do operador de importação *with* em um AADL Package privado.

Na Listagem 3.11 é mostrado um exemplo de declaração da função privada *função2* na linguagem C#. Em linguagens orientadas a objetos, uma função privada é acessível somente dentro do escopo da própria classe. Em AADL, declarações de funções com modificadores de acesso do tipo privado são mapeadas para declarações de funções dentro da declaração de um AADL *Package* com modificador de acesso *private* como mostrado na Listagem 3.12 (linha 2).

Listing 3.11: Função privada em C#

```
1  public class ClassExemplo {
2    private void funcao2(){
3
4    }
5  }
```

Listing 3.12: Função privada em AADL

```
1  package PacoteExemplo
2  public
3
4      process ClassExemplo
5      end ClassExemplo;
6
7      process implementation ClassExemplo.imp
8          subcomponents
9              F1: thread funcao2;
10     end ClassExemplo.imp;
11
12     thread funcao2
13     end funcao2;
14
15 private
16     with PacoteExemplo
17
```

```
18      thread implementetion funcao2.imp
19      end funcao2.imp;
20
21  end PacoteExemplo;
```

Os parâmetros de entrada da declaração de uma função (método) em C#, como mostrado na Listagem 3.13, podem ser mapeados para portas de entrada declaradas dentro do compartimento *features* de um componente do tipo *thread* em AADL. Dessa forma, a função2 com o parâmetro booleano "b", declarada dentro da ClasseExemplo na listagem 3.13 (linha 2), foi mapeada para a especificação do componente *thread* função2 que inclui a declaração de um parâmetro de entrada *in data port* do tipo booleano dentro do compartimento features da thread (linhas 13 à 16 da listagem 3.14).

Listing 3.13: Parametros em C#

```
1  public class ClassExemplo {
2      public void funcao2(boolean b){
3
4      }
5  }
```

Listing 3.14: Parâmetros em AAD

```
1  package PacoteExemplo
2  public
3      with Base_Types;
4
5       process ClassExemplo
6      end ClassExemplo;
7
8      process implementation ClassExemplo.imp
9          subcomponents
10             F1: thread funcao2;
11     end ClassExemplo.imp;
12
13     thread funcao2
14         features
15             b: in data port Base_Types::Boolean;
16     end funcao2;
17
18     thread implementation funcao2.imp
19     end funcao2.imp;
20 end PacoteExemplo;
```

As declarações de variáveis internas a uma função em linguagens como C# e Java, como mostrado na Listagem 3.15, podem ser mapeadas para declarações de com-

ponentes do tipo *data* como subcomponentes de uma thread AADL, como mostrado na listagem 3.16. Dessa forma, a declaração da variável interna "f"do tipo *float* dentro da função1 na listagem 3.15 é mapeada para a declaração de um *data componente* dentro da implementação da thread funcao1 (linhas 16 à 19 da listagem 3.16).

Listing 3.15: Variaveis internas em C#

```
1  public class ClassExemplo {
2      public void funcao1(){
3          float f;
4      }
5  }
```

Listing 3.16: Variaveis internas em AADL

```
1  package PacoteExemplo
2  public
3      with Base_Types;
4
5       process ClassExemplo
6      end ClassExemplo;
7
8      process implementation ClassExemplo.imp
9          subcomponents
10             F1: thread funcao1;
11     end ClassExemplo.imp;
12
13     thread funcao1
14     end funcao1;
15
16     thread implementation funcao1.imp
17      subcomponents
18           f: data Base_Types::Float;
19     end funcao1.imp;
20 end PacoteExemplo;
```

Chamadas de funções dentro de outras funções são mapeadas para chamadas à subprogramas em AADL. Em AADL, chamadas à subprogramas somente podem ser feitas em implementações de componentes do tipo *thread* dentro do compartimento *calls* de uma *thread implementation*. Dessa forma, a chamada da *funcao2* dentro da declaração da *função1* da ClasseExemplo mostrada na listagem 3.17 (linha 3) é mapeada em AADL para a chamada ao subprograma funcao2 especificada dentro do bloco calls de uma thread implementation como mostrado na listagem 3.18 (linha 19).

Listing 3.17: Chamada de função em C#

```
1  public class ClassExemplo {
2      public void funcao1(){
3          funcao2();
4      }
5  }
```

Listing 3.18: Chamada de função em AADL

```
1  package PacoteExemplo
2  public
3      with Base_Types;
4
5       process ClassExemplo
6      end ClassExemplo;
7
8      process implementation ClassExemplo.imp
9          subcomponents
10             F1: thread funcao1;
11             F2: subprogram funcao2;
12     end ClassExemplo.imp;
13
14     thread funcao1
15     end funcao1;
16
17     thread implementation funcao1.imp
18      calls myCall: {
19          F2: subprogram funcao2.imp;
20      };
21     end funcao1.imp;
22
23     thread funcao2
24     end funcao2;
25
26     subprogram implementation funcao2.imp
27     end funcao2.imp;
28 end PacoteExemplo;
```

Passagens de parâmetros em linguagens como C# e Java, como mostrado na listagem 3.19 (linhas 3 e 5), são mapeados em AADL para especificações de conexões entre parâmetros de uma thread e parâmetros de um subprograma dentro do bloco *connections* de um *thread implementation* como mostrado na listagem 3.20 (linhas 23 e 24).

Listing 3.19: Passagem por parâmetros em C#

```
1  public class ClassExemplo {
2      public void funcao1(){
3          float f;
4
5          funcao2(f);
```

```
6     }
7 }
```

Listing 3.20: Passagem por parâmetros em AADL

```
1  package PacoteExemplo
2  public
3      with Base_Types;
4
5       process ClassExemplo
6      end ClassExemplo;
7
8      process implementation ClassExemplo.imp
9          subcomponents
10             F1: thread funcao1;
11             F2: subprogram funcao2;
12     end ClassExemplo.imp;
13
14     thread funcao1
15         features
16             f: out data port Base_Types::Float;
17     end funcao1;
18
19     thread implementation funcao1.imp
20      calls myCall: {
21          F2: subprogram funcao2.imp;
22      };
23      connections
24          C0: parameter f -> F2.b;
25     end funcao1.imp;
26
27     subprogram funcao2
28         features
29             b: in parameter Base_Types::Float;
30     end funcao2;
31
32     subprogram implementation funcao2.imp
33     end funcao2.imp;
34 end PacoteExemplo;
```

O retorno de uma função em C#, como mostrado na listagem 3.21 (linha 3), também pode ser mapeado em AADL para a especificação de uma conexão entre o parâmetro de uma thread e o parâmetro de um subprograma dentro do bloco *connections* de um *thread implementation* (linha 24 da listagem 3.22).

Listing 3.21: Retorno em C#

```
1 public class ClassExemplo {
2     public void funcao1(){
3         r = funcao2();
```

```
4     }
5 }
```

Listing 3.22: Retorno em AADL

```
1  package PacoteExemplo
2  public
3      with Base_Types;
4
5       process ClassExemplo
6      end ClassExemplo;
7
8      process implementation ClassExemplo.imp
9          subcomponents
10             F1: thread funcao1;
11             F2: subprogram funcao2;
12     end ClassExemplo.imp;
13
14     thread funcao1
15         features
16             f: in data port Base_Types::Float;
17     end funcao1;
18
19     thread implementation funcao1.imp
20      calls myCall: {
21          F2: subprogram funcao2.imp;
22      };
23      connections
24          C0: parameter F2.b -> f;
25     end funcao1.imp;
26
27     subprogram funcao2
28         features
29             b: out parameter Base_Types::Float;
30     end funcao2;
31
32     subprogram implementation funcao2.imp
33     end funcao2.imp;
34 end PacoteExemplo;
```

3.1.6 Mapeamento de Polimorfismo

AADL não suporta o conceito de polimorfismo de linguagens de programação orientadas a objetos. Dessa forma, cada método polimórifico declarado em uma classe C# ou Java, como ilustrado na listagem 3.23, pode ser mapeado para um componente AADL do tipo thread, desde que cada componente que represente um método polimórfico possua um nome diferente. Na listagem 3.24, os métodos polimórficos funcao() e funcao(int i) foram

mapeados para as threads funcao1 (linhas 14 a 18) e funcao2 (linhas 20 a 26).

Listing 3.23: Polimorfismo em C#

```
1 public class ClassExemplo {
2     public void funcao(){
3
4     }
5
6     public void funcao(int i){
7
8     }
9 }
```

Listing 3.24: Polimorfismo em AADL

```
1  package PacoteExemplo
2  public
3      with Base_Types;
4
5       process ClassExemplo
6      end ClassExemplo;
7
8      process implementation ClassExemplo.imp
9          subcomponents
10             F1: thread funcao1;
11             F2: thread funcao2;
12     end ClassExemplo.imp;
13
14     thread funcao1
15     end funcao1;
16
17     thread implementation funcao1.imp
18     end funcao1.imp;
19
20     thread funcao2
21         features
22             i: in data port Base_Types::Integer;
23     end funcao2;
24
25     thread implementation funcao2.imp
26     end funcao2.imp;
27 end PacoteExemplo;
```

3.1.7 Mapeamento de Relações do Tipo Generalização/Especialização entre Objetos

Da mesma forma que linguagens orientadas a objetos como C# e Java, AADL também suporte o conceito de herança de objetos. Para tanto, basta utilizar o comando *extends*

na especificação de componentes AADL. Na listagem 3.25 é mostrado um exemplo de relacionamento de herança em que Class2 estende dados e comportamentos da classe ClasseExemplo. Essa relação é mapeada para a declaração do componente *Class2* do tipo *process* que estende a declaração do componente *ClasseExemplo* também do tipo *process* utilizando o comando *extends* de AADL.

Listing 3.25: Herança em C#

```
1  public class ClassExemplo {
2
3  }
4
5  public class Class2 extends ClassExemplo{
6
7  }
```

Listing 3.26: Herança em AADL

```
1  package PacoteExemplo
2  public
3      process ClassExemplo
4      end ClassExemplo;
5
6      process implementation ClassExemplo.imp
7      end ClassExemplo.imp;
8
9      process Class2 extends ClassExemplo
10     end Class2;
11
12     process implementation Class2.imp
13     end Class2.imp;
14 end PacoteExemplo;
```

3.1.8 Mapeamento de Classes Abstratas

O conceito de classes abstratas é outra característica de linguagens orientadas a objetos que também é suportado por AADL. Da mesma forma que linguagens orientadas a objeto como C# e Java, AADL também possui o modificador *abstract* que pode ser utilizado para especificar *packages*, componentes do tipo *system*, hardware e de software como abstratos. Na listagem 3.27 é mostrada a declaração de uma classe abstrata em C#. Essa declaração é mapeada em AADL para a declaração um *package* abstrato.

Listing 3.27: Classe abstrata em C#

```
1 public abstract class ClassExemplo {
2
3 }
```

Listing 3.28: *Abstract* em AADL

```
1 package PacoteExemplo
2 public
3     abstract process ClassExemplo
4     end ClassExemplo;
5
6     abstract process implementation ClassExemplo.imp
7     end ClassExemplo.imp;
8
9 end PacoteExemplo;
```

3.2 Abordagem Proposta

Nesta seção é apresentada a abordagem proposta para apoiar o processo de reengenharia de código fonte para modelos AADL. A abordagem proposta foi elaborada com base nos mapeamentos entre código fonte e representações AADL apresentados na Seção 3.1.

A realização do processo de reengenharia proposto foi realizado com o auxilio da ferramenta EPF Composer citada na Seção 2.7.2 no Capítulo 2.

Primeiramente, é necessário identificar os pacotes de um projeto C# ou Java e mapeá-los para AADL *packages*. Dessa forma, para cada pacote, cria-se um AADL *package*. Posteriormente, é necessário identificar as Classes que estão dentro de cada pacote C# e mapeá-las para componentes de software em AADL. Na Figura 3.1 é mostrado o fragmento do Diagrama de Atividades SPEM 2.0 da abordagem de reengenharia proposta que detalha o processo de identificação dos pacotes e das classes e as transformações/mapeamentos desses fragmentos de código fonte para as suas respetivas representações em AADL.

Para cada classe identificada dentro de um pacote de um projeto C# ou Java, deve-se declarar um *process* com o nome da classe e um *process implementation* dentro do *package* apropriado. Posteriormente, cada classe deve ser analisada para identificar as bibliotecas e as dependências da classe sob análise em relação às outras classes. Dessa

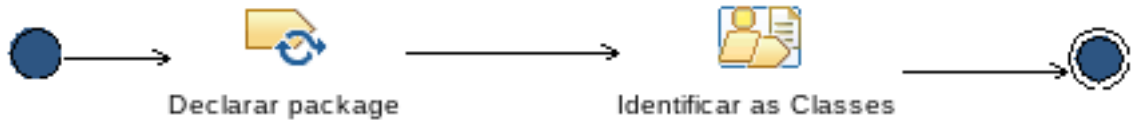

Figura 3.1: Diagrama de atividades SPEM 2.0 para identificação de pacotes e classes

forma, para cada dependência identificada declarar essa dependência utilizando o operador *with* dentro de um AADL *package* no qual o processo que representa a classe está contido. Ainda dentro da classe, é necessário identificar os tipos de dados por referência declarado no escopo da classe em análise. Para cada tipo de dado identificado, declarar um *data* com o nome do novo tipo de dado. Para cada classe sob análise, também é necessário identificar as variáveis globais da mesma. Cada variável global identificada deve ser declarada como portas de entrada e/ou de saída dentro do compartimento *features* do *process* componente. Dentro da classe sob análise, também é necessário identificar as funções/métodos da classe. Para cada função, é necessário declarar uma *thread* e uma *thread implementation* com o nome da função. O diagrama de atividade SPEM 2.0 para esse processo é mostrado na Figura 3.2.

Para cada função identificada, deve-se declarar uma *thread* dentro do *subcomponents* do *process implementation*. Ainda para cada função, deve-se identificar se a função é privada ou pública. Se for privada a declaração deve ser dentro do escopo *private* e se for pública dentro do *public*. Dentro de cada função identificar se a mesma possui variáveis internas, se a variável for passada por parâmetro para outra função ou o retorno de uma função, deve-se declará-la como *out data port* dentro da *features* da *thread* com o nome da variável e o tipo. Se a variável for para receber retorno de outra função declara-se a variável como *in data port* no mesmo lugar em que foi declarado as outras variáveis internas com o nome e o tipo da variável. A declaração de variáveis internas é representado no diagrama SPEM 2.0 da Figura 3.3.

Se a função identificada possuir parâmetros de entrada, deve-se declará-los como *in data port* dentro das *features* da *thread* com o nome e o tipo da variável. Para cada função, deve-se identificar se a função possui chamadas a outras funções. Os passos para realizar tal análise são mostrados diagrama SPEM 2.0 da Figura 3.4.

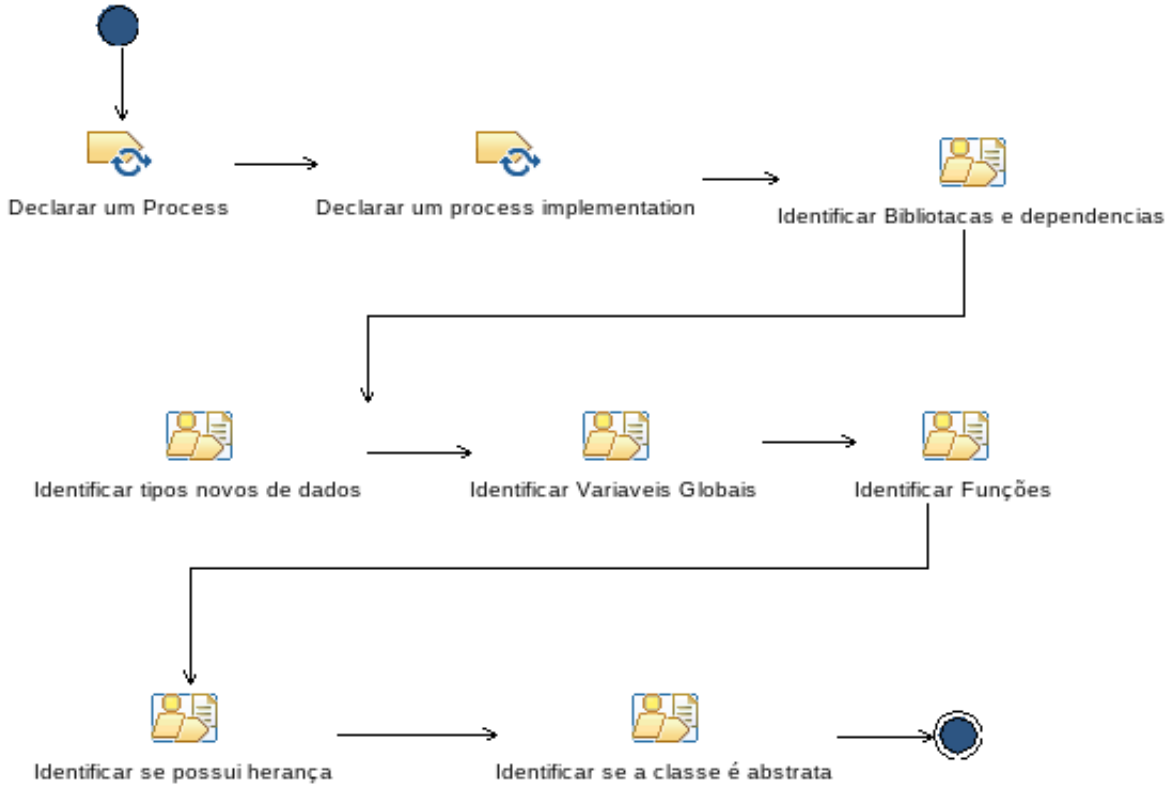

Figura 3.2: Diagrama SPEM 2.0 para transformar classes em elementos AADL.

Cada chamada de função identificada, deve ser declarada dentro do compartimento *calls* de um *thread implementation*, mas a declaração da função deve ser feita dentro de um *thread implementation* e a função chamada deve mudar o tipo de *thread* para *subprogram*. Isso é necessário pois uma *thread* não possui chamada para outra *thread*, somente para um *subprogram*. Se a função chamada possuir passagens por parâmetros, essas passagens devem ser declaradas dentro de compartimentos *connections* de *thread implementations* ou *subprogram implentations*. Dependendo de como foi declarada a função, atribui-se um nome para essa conexão declarando-a como *parameter* e informando a variável que vai passar e a variável que vai receber o retorno da função. Se a função chamada possuir retorno deve-se declarar dentro da *connections*, assim como fez na passagem por parâmetro informando a variável que vai ser retornada e a variável que vai receber o retorno. Na Figura 3.5 é mostrado o diagrama SPEM 2.0 que descreve o processo de transformação de chamadas de funções para representações em AADL.

Caso uma classe herde de outra classe, basta utilizar o comando *extends*, da mesma forma que em linguagens de programação orientadas a objetos como C# e Java. Se a classe for abstrata, deve-se trocar a declaração de *process* para *abstract*. O mesmo é

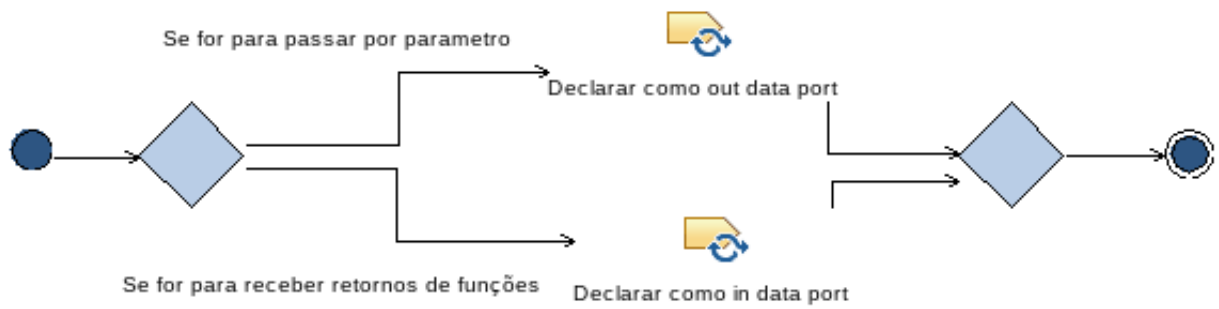

Figura 3.3: Diagrama de atividades SPEM 2.0 para mapear variáveis internas em representações AADL.

válido para uma função ou variável. Uma visão geral do processo de reengenharia proposto é mostrado na Figura 3.6, onde o processo se inicia identificando os pacotes dentro de um projeto C#.

Para cada pacote identificado é realizado uma iteração, em cada iteração do pacote deve-se declarar esse pacote como *package* e identificar as classes desse pacote.

Para cada classe identificada realizar uma iteração declarando a classe como *process* e sua implementação, identificar as bibliotecas e dependências declarando como *with*; identificar tipos novos de dados e declara-los como *data*; identificar variáveis globais e declará-las como *in out data port*; identificar as funções de cada classe.

Para cada função identificada realiza-se uma iteração declarar a função do tipo *thread* e sua implementação na parte pública se a função for pública ou na parte privada se a função for privada; identificar se a função possui variáveis internas e declara-las como *in out data port*; identificar se a função possui parâmetros de entrada e delcarar como *in data port*; identificar se a função possui chamada de outras funções, para cada chamada de função declará-la como *subprogram* e fazer as alterações necessárias; identificar se a função chamada possui passagem por parâmetro e para cada passagem por parâmetro declarar a *connections*; identificar se a função chamada possui retorno e declarar a *connections*, se a função identificada for abstrata deve-se alterar o tipo de componente para *abstract*, se a classe identificada possui herança declarar a herança como *extends* e se a classe for abstrata alterar o tipo de componente para *abstract*.

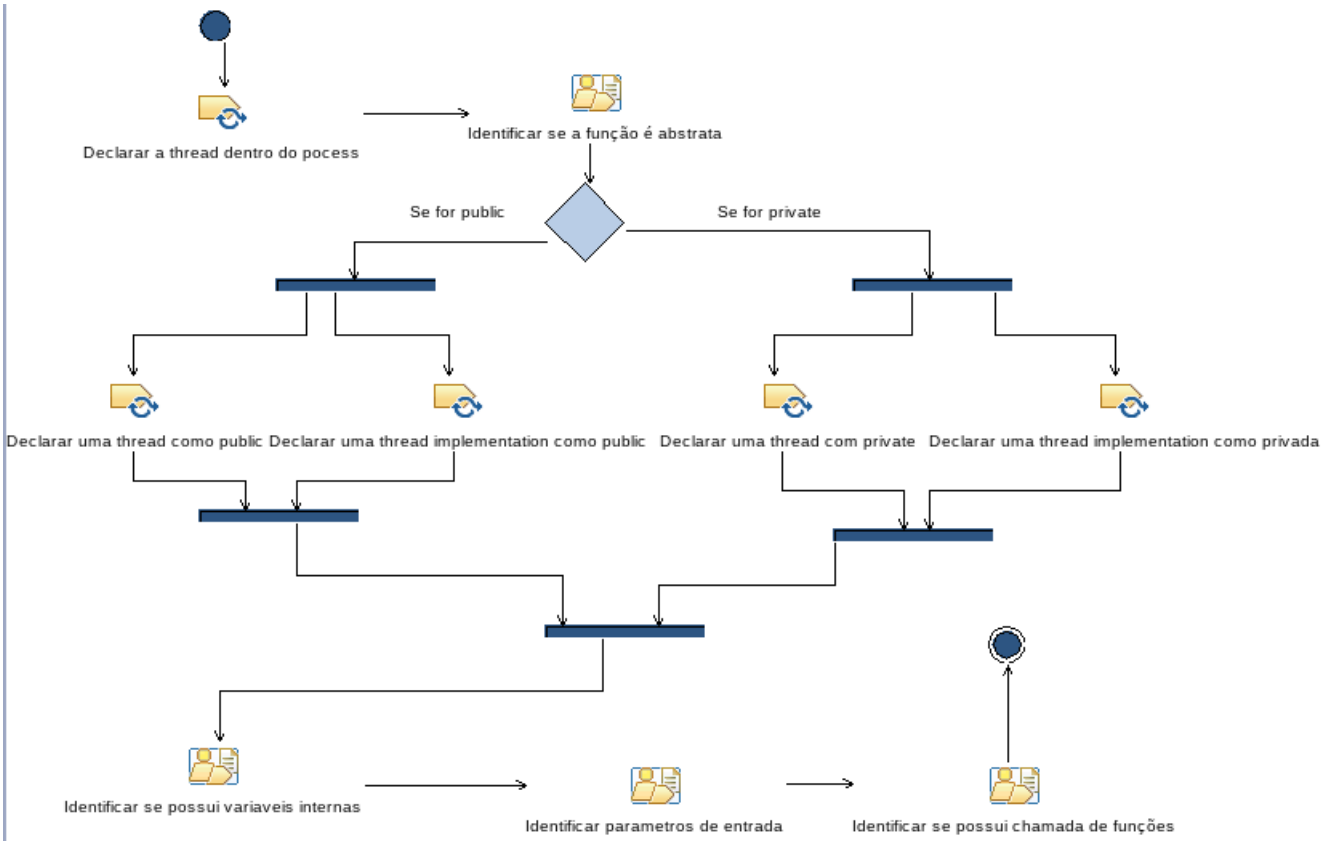

Figura 3.4: Diagrama de atividades SPEM 2.0 para mapear funções para representações AADL.

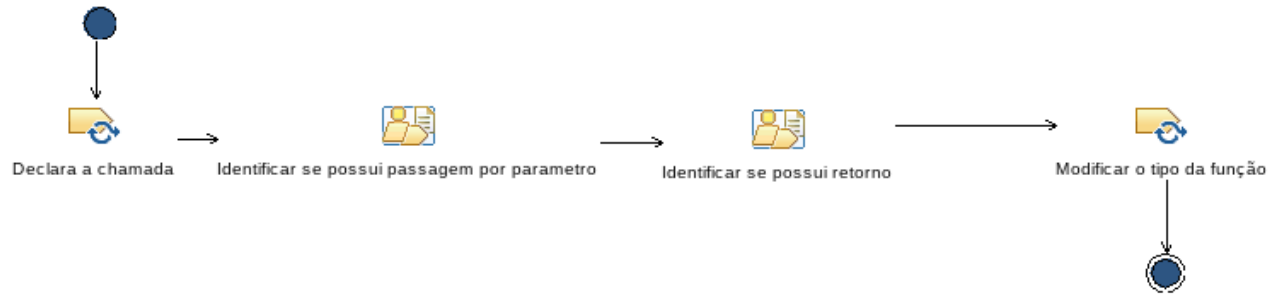

Figura 3.5: Diagrama de atividades SPEM 2.0 para mapear chamadas de funções para representações em AADL.

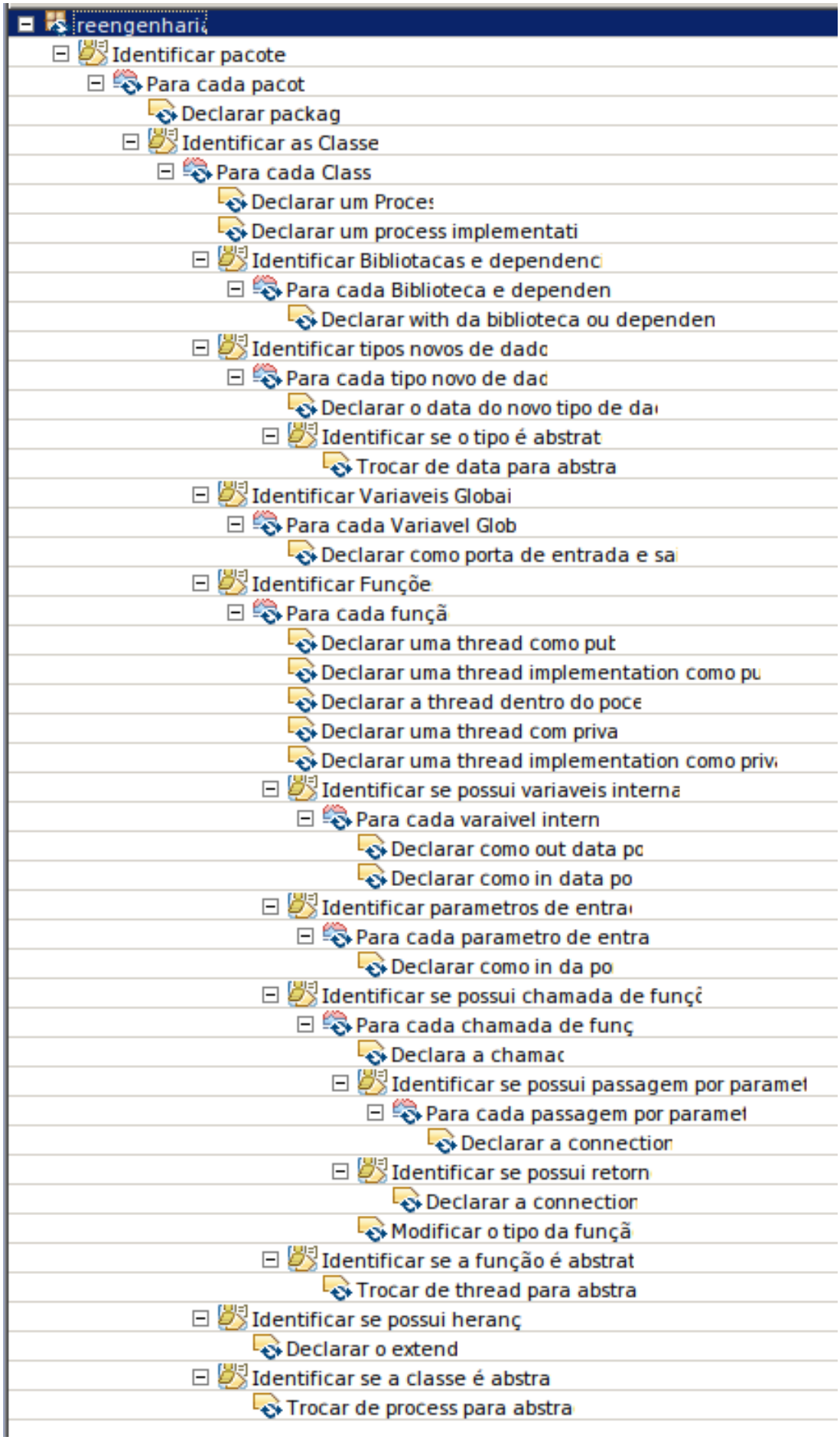

Figura 3.6: Visão geral da abordagem proposta.

4 Estudo de Caso

Neste capítulo é apresentado um estudo de caso da aplicação da abordagem proposta na reengenharia de parte do sistema controlador de voo ArduPilot implementado em C# para a sua respectiva representação como modelos AADL.

4.1 ArduPilot

ArduPilot é um software livre para controle de piloto automático. ArduPilot pode ser utilizado para controlar drones, aeronaves de asa fixa e VTOL, helicópteros, ground rovers, barcos, submarinos e rastreadores de antena. Inicialmente foi desenvolvido para controlar aeronaves e robôs e evoluiu para piloto automático. ArduPilot é um software de navegação juntamente com um software de controle de estação como o Mission Planner, o APM Planner, o QGroundControl, o MavProxy, o Tower. O código do ArduPilot está disponível no GitHub (MISSIONPLANNER,). O software de código aberto está constantemente sendo atualizado por uma equipe de 30 desenvolvedores principais, apoiada por uma comunidade de mais de 10.000 membros. (ARDUPILOT MEGA,)

4.2 Aplicação da Abordagem de Reengenharia Proposta

Nesta seção é apresentada a aplicação da abordagem proposta no processo de reengenharia do sistema ArduPilot(MISSIONPLANNER,). O código fonte do ArduPilot está disponível, mas não possui todas as classes, portanto o mapeamento de alguns métodos foram inferidos e declarados como componentes AADL do tipo *data*. O processo de reengenharia proposto foi aplicado a uma porção do código fonte do ArduPilot, que inclui a classe *AuthKeys* do pacote *controls* e a classe *OpticalFlow* do pacote *utilities*. Nas seções subsequentes são apresentadas as transformações de fragmentos de código fonte dessas classes para as suas respectivas representações em AADL seguindo os passos da

abordagem de reengenharia proposta no Capítulo 3 deste trabalho de conclusão de curso.

4.2.1 Identificação de pacotes e mapeamentos para representações AADL

As classes *AuthKeys* e *OpticalFlow* do sistema ArduPilot estão respectivamente armazenadas nos pacotes *controls* e *utilities* como mostrado na Listagem 4.1. No processo de reengenharia do sistema ArduPilot para modelos AADL, dois elementos do tipo AADL *Package* foram criados para representar os pacotes *controls* e *utilities* (Listagem 4.2). De forma análoga a um pacote de um projeto em C# que armazena um conjunto de classes, cada AADL *Package* representa um *namespace* para componentes, que podem incluir tanto componentes de hardware como *memory*, *processors* e *buses process* e *thread*, quanto componentes do tipo *system* e componentes de software como *process* e *thread*.

Listing 4.1: Namaspaces em C#

```
1  namespace MissionPlanner.Controls{
2      public partial class AuthKeys : Form {...}
3      ...
4  }
5  namespace MissionPlanner.Utilities{
6      public class OpticalFlow: IDisposable {
7          MAVLinkInterface  mav;
8      ...
9      }
10     ...
11 }
```

Listing 4.2: Representações de Namaspaces em C# como Packages em AADL

```
1  package Controls
2      with Mavlink, Utilities,...;
3  end Controls;
4
5  package Utilities
6      with System;
7  end Utilities;
```

4.2.2 Identificação de classes e seus mapeamentos para representações AADL

Após a identificação de todos os pacotes de um projeto C#, C++ ou Java e mapeá-los para elementos do tipo *Package* em uma especificação AADL, deve-se analisar as classes contidas em cada pacote e mapeá-las para componentes de software em AADL como *process* e *thread*. Na Listagem 4.1 é mostrado que os pacotes *controls* e *utilities* do sistema ArduPilot contêm respectivamente as classes *AuthKeys* e *OpticalFlow*. Cada uma dessas classes foi mapeada para um elemento AADL do tipo *process*. Um componente do tipo *process* representa um espaço de endereçamento de forma análoga aos conceitos de classe e objeto de linguagens de programação orientadas a objetos. Como quaisquer componentes AADL, a *especificação* de um componente do tipo *process* contém a *definição* do componente, que inclui a especificação das portas de entrada e de saída do componente declaradas dentro do bloco *features*, e a *implementação* do componente, que inclui a especificação de subcomponentes e conexões entre portas e subcomponentes. Na listagem 4.3 é mostrado o mapeamento da classe *AuthKeys* para uma representação vazia de um componente de software AADL do tipo *process*. Isso se justifica pelo fato da classe *AuthKeys* não possuir declarações de variáveis globais. As variáveis globais de uma classe C#, como *OpticalFlow* podem ser mapeadas para declarações de portas, e.g., do tipo *inout* dentro do bloco *features* de um componente AADL do tipo *process*, como mostrado na representação da classe *OpticalFlow* como um component AADL do tipo *process* que contém a declaração da porta *mav* do tipo *inout* dentro do bloco *features* na Listagem 4.4.

Listing 4.3: Representação da classe AuthKeys

```
1 package Controls
2     with Form, Utilities;
3     process AuthKeys extends Form
4     end AuthKeys;
5
6     process implementation AuthKeys.impl
7     end AuthKeys.impl;
8 end Controls;
```

Listing 4.4: Representação da classe OpticalFlow como um AADL

```
1  package Utilities
2      with System;
3      process OpticalFlow extends IDisposable
4          features
5              mav: in out data port MAVLinkInterface;
6      end OpticalFlow;
7
8      process implementation OpticalFlow.impl
9      end OpticalFlow.impl;
10 end Utilities;
```

Identificação de relações generalização/especialização entre classes e seus mapeamentos para representações AADL

Após a identificação das classes de um projeto C# e mapeá-las para elementos do tipo *process* em uma especificação AADL, deve-se analisar se as classes identificadas possuem relações de generalização/especialização com outras classes. Uma vez que AADL possui suporte para a especificação de relações generalização/especialização entre os seus elementos, as relações generalização/especialização de linguagens de programação orientadas a objetos como C# e Java, devem ser mapeadas para relações generalização/especialização entre componentes AADL expressas por meio da seguinte sintaxe *Elemento1 AADL extends Elemento2*.

Na linguagem C#, uma relação generalização/especialização entre classes é representada por *:* como mostrado na Listagem 4.1. A classe *AuthKeys* é uma especialização da classe *Form* e classe *OpticalFlow* é uma especialização da classe/interface *IDisposable*, como denotado por *:*. As relações de generalização/especialização entre *AuthKeys*/*Form* e *OpticalFlow*/*IDisposable* foram mapeadas para relações generalização/especialização entre elementos AADL utilizando a palavra reservada *extends* como mostrado nas Listagens 4.3 (linha 3) e 4.4 (linha 3).

4.2.3 Identificação de bibliotecas e relacionamentos de dependência entre classes e seus mapeamentos para representações AADL

Após identificar todas as classes de um projeto C# e mapeá-las para componentes AADL do tipo *process*, deve-se analisar cada classe e identificar se possui referências à outras

classes, expressa por declarações de uso e/ou importação de classes armazenadas em biblioteca de código fonte. Declarações de importação de classes armazenadas em bibliotecas de código em classe C#, C++ ou Java devem ser mapeadas para declarações de importação de pacotes AADL, que representam biblioteca de modelos, dentro de um elemento do tipo *AADL Package* utilizando o comando de importação *with* como mostrado na Listagem 4.3 (linha 2). Neste exemplo, o comando *with* indica que o AADL Package *Controls* referencia e utiliza elementos AADL declarados no escopo do pacote *Utilities*. As declarações de importações dentro de uma classe são feitas utilizando o comando *using* em C# e *import* na linguagem Java. O comando/operador *with* é análogo aos conceitos de *import* da linguagem Java e *using* em C#.

4.2.4 Identificação de tipos de dados por referência declarados em classes e seus mapeamentos para representações AADL

Após a identificação de todas as classes de um projeto C# e mapeá-los para componentes em AADL como *process*, cada classe deve ser analisada para identificar se a mesma possui variáveis globais de tipos por referência. Para cada tipo de dado por referência identificado, deve-se declarar esse novo tipo de dado em AADL como um componente *data*. Na listagem 4.5 é mostrado que a classe *OpticalFlow* possui uma variável global do tipo *MAVLinkInterface*, que é um tipo de dado por referência. MAVLinkInteraface foi mapeado para um componente AADL do tipo *data*, como mostrado na Listagem 4.6 (linhas 3 e 4).

Listing 4.5: Declaração de um tipo de dado por referência na classe OpticalFlow

```
1 namespace MissionPlanner.Utilities{
2     public class OpticalFlow{
3         MAVLinkInterface mav;
4     }
5 }
```

Listing 4.6: Representação de um tipo de dado por referência na classe OpticalFlow em um componente AADL do tipo process

```
1 package Utilities
2 public
3     data MAVLinkInterface
```

```
4           ...
5       end MAVLinkInterface;
6
7       process OpticalFlow extends IDisposable
8           features
9               mav: in out data port MAVLinkInterface;
10      end OpticalFlow;
11  end Utilities;
```

4.2.5 Identificação de variáveis globais declaradas em classes C# e seus mapeamentos para representações AADL

Após a identificação de todas as classes de um projeto C# e mapeá-las para componentes em AADL como *process*, cada uma dessas classes foram analisadas para identificar declarações de variáveis globais e mapeá-las para declarações de portas (*in*, *out*, ou *in/out*), em componentes AADL do tipo *process*, *thread*, *data* ou *subprogram*. Na Listagem 4.5 é mostrada a declaração da variável global *mav* do tipo *MAVLinkInterface* na classe *OpticalFlow* (linha 3). Na Listagem 4.6 é mostrado o mapeamento dessa declaração de variável global para um elemento porta do tipo *in/out* dentro do compartimento *features* do processo *OpticalFlow* (linha 9). Declarações de elementos do tipo porta devem ser feitas dentro do compartimento *features* de componentes AADL do tipo *process*, *subprogram*, *thread* ou *data*.

4.2.6 Identificação de funções/métodos definidos em classes e seus mapeamentos para representações AADL

Nesta etapa, o código fonte de cada uma das classes C# do sistema ArduPilot foi analisado e cada função/método identificado foi mapeado em AADL para um componente de software do tipo *thread*. Na Listagem 4.7 é mostrada a declaração da função *but save click* da classe *AuthKeys* na linguagem C# (linhas 3 a 6). A declaração de uma função em C# inclui a especificação do modificador de acesso, tipo de retorno, assinatura e lista de parâmetros (Listagem 4.7). A função da Listagem 4.7 foi mapeada para uma declaração e implementação de componente AADL do tipo *thread* (linhas 8 e 9 e linhas 10 a 17 da Listagem 4.8) e para a declaração do subcomponente *BSC* do tipo *butsaveclick.impl* do

componente AADL do tipo *process AuthKeys* (linha 5). Dessa forma, um componente AADL do tipo *thread* foi criado para cada função declarada na classe *AuthKeys* do sistema ArduPilot. A implementação de cada *thread* é referenciada como um subcomponente do componente AADL do tipo *process AuthKeys*, como mostrado na Listagem 4.8. Assim, para cada função mapeada para um componente do tipo *thread*, é necessário declarar a implementação desta *thread* dentro do bloco *subcomponents* do componente AADL *AuthKeys* do tipo *process* que a função pertence (linha 5 da Listagem 4.8). Pode-se observar, que na listagem 4.7, a função *but_save_click* é declarada privada (linha 3). Por essa razão, esta função foi mapeada para uma declaração e uma implementação de um componente AADL do tipo *thread* com o modificador de acesso privado (*private*) (linhas 8 e 11 da Listagem 4.8).

Listing 4.7: Declaração da função but_save_click da classe AuthKeys na linguagem C#

```
1 namespace MissionPlanner.Controls{
2     public partial class AuthKeys : Form{
3         private void but_save_Click(object sender, EventArgs e){
4             Save();
5         }
6     }
7 }
```

Listing 4.8: Representação de uma função da classe *AuthKeys* em AADL

```
1  package Controls
2  public
3      process implementation AuthKeys.impl
4          subcomponents
5              BSC: thread but_save_Click.impl;
6      end OpticalFlow;
7
8      private  thread but_save_Click
9      end but_save_Click;
10
11     private
12         thread implementation but_save_Click.impl
13     end but_save_Click.imp;
14 end Utilities;
```

4.2.7 Mapeamento de variáveis internas declaradas no escopo funções/métodos para representações AADL

Após a identificação de todas as funções de uma classe C# e mapeá-las para componentes AADL do tipo *thread*, cada função foi analisada para verificar se a mesma possui a declaração de variáveis internas e mapeá-las para declarações de sub-componentes do tipo *data* em uma implementação de uma *thread* em AADL para o tipo *data* dentro do atributo. Na Listagem 4.9 é mostrada a declaração da função *but_add_click* da classe *AuthKeys*, que inclui duas variáveis internas (linhas 4 e 5), sendo uma variável do tipo *int* e outra do tipo *String*. As declarações dessas variáveis internas foram mapeadas em AADL para declarações de dois sub-componentes do tipo *data* em uma implementação da *thread but_add_click*, como mostrado na Listagem 4.10 (linhas 5 e 6).

Listing 4.9: Variáveis internas em C#

```
1 namespace MissionPlanner.Controls{
2     public partial class AuthKeys : Form{
3         private void but_add_Click(Object sender, EventArgs e){
4             int row;
5             String name;
6         }
7     }
8 }
```

Listing 4.10: Representações de variáveis internas de uma função do ArduPilot em AADL

```
1 package Controls
2 private
3     thread implementation but_add_Click.imp
4         subcomponents
5             row: data Base_Types::Integer;
6             name: data Base_Types::String;
7     end but_add_Click.imp;
8 end Utilities;
```

4.2.8 Mapeamento de parâmetros de uma função/método C# para representações AADL

Nesta etapa, os parâmetros entrada declaradas na lista de argumentos de uma função foram mapeados para declarações de portas dentro do compartimento *features* na definição

de um componente AADL do tipo *thread*. Na Listagem 4.9 é mostrada a declaração da função *but_add_click* que dois parâmetros: *sender* do tipo *Object* e *e* do tipo *EventArgs* (linha 3). Esses dois parâmetros foram mapeados para declarações de duas portas de entrada, uma do tipo *Object* e outra do tipo *EventArgs*, dentro do compartimento *features* da *thread* mostrada na Listagem 4.11 (linhas 4 à 6).

Listing 4.11: Representações de variáveis internas de uma função do ArduPilot em AADL

```
1 package Controls
2 public
3     thread but_add_Click
4         features
5             sender: in data port object;
6             e: in data port EventArgs;
7     end but_add_Click;
8 end Utilities;
```

Mapeamento de chamadas de funções no código fonte para subprogramas AADL

Nesta etapa, cada função identificada foi analisada para verificar se a mesma possui chamadas de funções para mapeá-las para declarações de componentes AADL do tipo *subprogram* e chamadas desses subprogramas em implementações de componentes do tipo *thread* dentro do compartimento *calls*. As chamadas de funções dentro de uma outra função foram mapeadas para componentes do tipo *subprogram*, que posteriormente foram declarados dentro compartimentos *calls* de implementações de componentes AADL do tipo *thread*. Isso foi feito pois uma *thread* não pode conter camadas para componentes do tipo *thread*, somente para componentes do tipo *subprogram*. Essas modificações também foram feitas em declarações dentro do compartimento *features* de componentes AADL do tipo *process*, nos quais declarações de variáveis internas à essas funções foram mapeadas para declarações de elementos do tipo *parameter* ao invés de elementos do tipo do tipo *data port*. Na Listagem 4.7 é mostrada a chamada da função *save* dentro da função *but save click* (linha 4) da classe *AuthKeys*. Na Listagem 4.12 é mostrado o mapeamento da chamada da função *save* para uma chamada (linhas 4 a 6) ao *subprogram save* (linha 9 a 10) dentro do compartimento *calls* da implementação do componente AADL *butsaveclick* do tipo *thread*.

Listing 4.12: Representação de chamada da função save dentro da função but_save_click da classe *AuthKeys* em AADL

```
1  package Controls
2  private
3      thread implementation but_save_Click.imp
4          calls MyCall: {
5              S: subprogram Save;
6          };
7      end but_save_Click.imp;
8
9      subprogram Save
10     end Save;
11 end Controls;
```

4.2.9 Mapeamento de passagem por parâmetro no código fonte C# para subprogramas AADL

Nesta etapa, as chamadas de função foram analisadas para verificar a existência de passagem por parâmetros entre funções, para posteriormente mapeá-las em AADL para conexões (*connections*) dentro de implementação de componentes AADL dos tipos *thread* ou *subprogram*. Neste mapeamento, para cada passagem de parâmetros entre funções, foram indicadas as variáveis de origem e de destino. Na Listagem 4.13 é mostrado um exemplo de passagem por parâmetro entre funções no qual a variável *bmp* do tipo *MAVLinkMessage* declarada como parâmetro da função *ReceivedPackage* da classe *OpticalFlow* é passado para a chamada da função *SetGrayscalePalette* dentro da função *ReceviedPackage* (linha 5). O mapeamento da passagem de parâmetro entre essas duas funções para AADL é mostrado na Listagem 4.14, no qual a variável passada por parâmetro foi declarada na linha 5 e a variável destino foi declarada como parâmetro de entrada da função destino *SetGrayscalePalette* dentro do compartimento *features* de um subprograma AADL (linha 15).

Listing 4.13: Passagem de parâmetro em C#

```
1  namespace MissionPlanner.Utilities{
2      public class OpticalFlow: IDisposable{
3          private bool ReceviedPackage(MAVLink.MAVLinkMessage bmp){
4              ...
5              SetGrayscalePalette(bmp);
6              var bitmapData = LockBits();
7              ...
```

```
8           }
9       }
10  }
```

Listing 4.14: Representações de passagem de parâmetro entre chamadas de funções da classe *AuthKeys* em AADL

```
1  package Utilities
2  private
3      thread implementation ReceviedPackage.imp
4          subcomponents
5              bmp: out data port Bitmap;
6          calls MyCall: {
7              SGP: subprogram SetGrayscalePalette;
8          };
9          connections
10             C0: parameter bmp -> SGP.bmp;
11     end ReceviedPackage.imp;
12
13     subprogram SetGrayscalePalette
14         features
15             bmp: in parameter Bitmap;
16     end SetGrayscalePalette;
17 end Utilities;
```

4.2.10 Mapeamento de retorno de funções no código fonte C# para subprogramas AADL

Após identificar todas as chamadas de funções de um projeto C# e mapeá-los para AADL, deve-se analisar as chamadas de funções e identificar se possui retorno de parâmetro e mapeá-lo como conexão (*connections*) dentro da parte de implementação de *thread* ou *subprogram*. Na Listagem 4.13 é mostrada a variável *bitmapData* recebendo o retorno da função *LockBits* dentro da função *ReceviedPackage* (linha 6). O mapeamento do retorno de uma função é mostrado na Listagem 4.15, onde a variável destino foi declarada na linha 5 e a variável de origem foi declarada dentro da função que possui o retorno (linha 15).

Listing 4.15: Representações de retornos de funções do ArduPilot em AADL

```
1  package Utilities
2  private
3      thread implementation ReceviedPacket.imp
4          subcomponents
```

```
5               bitmapData: in data port Base_Types::Unsigned_32;
6           calls MyCall: {
7               LK: subprogram LockBits;
8           };
9           connections
10              C0: parameter LK.bitmapData -> bitmapData;
11      end ReceviedPacket.imp;
12
13      subprogram LockBits
14          features
15              bitmapData: out parameter Base_Types::Unsigned_32;
16      end LockBits;
17  end Utilities;
```

5 Conclusões

A realização de atividades de Engenharia de Segurança a partir do código fonte de um sistema crítico previamente desenvolvido é um processo custoso e demorado, pois demanda a reengenharia do código fonte para modelos. Neste trabalho foi proposto uma abordagem dirigida a modelos para apoiar a reengenharia do código fonte de sistemas crítico. Na abordagem proposta foram formalizados mapeamentos entre abstrações de linguagens orientadas a objetos e representações equivalentes na linguagem AADL (Seção 3.1) e um conjunto de passos, na forma de uma abordagem (Seção 3.2).

Com a criação da abordagem proposta de mapeamento para AADL, tem-se grande apoio aos engenheiros de sistemas no processo de reengenharia do código fonte para especificações equivalentes em AADL, para posteriormente aplicar a engenharia de segurança no sistema.

A falta de um apoio ferramental fio um grande dificultador para a realização deste trabalho de conclusão de curso. A carência de abordagens que apoiam a reengenharia de código fonte para modelos também foi um fator limitante para o desenvolvimento do trabalho.

A abordagem proposta neste trabalho está em estágio preliminar e necessita de validação em outros estudos de caso. Também há a necessidade de desenvolvimento de ferramentas com suporte à transformações de modelos que permitam a conversão automática de código fonte de sistema implementados em linguagens de programação orientadas a objetos como C++, C# ou Java em representações equivalentes em AADL.

Como sugestões de trabalhos futuros estão o refinamento da abordagem proposta para apoiar outras linguagens de programação e a realização de estudos de casos em outros domínios como automotivo e aeroespacial para validar tanto a abordagem quanto o apoio ferramental ser desenvolvido. Além disso, para auxiliar a reengenharia de um código fonte para a sua especificação equivalente em AADL, também sugere-se como trabalhos futuros a criação de um plugin para a transformação de código fonte para representações AADL.

Bibliografia

ANDRADE, R. M. C. Sistemas críticos. In: . [s.n.], 2007. Disponível em: ⟨http://disciplinas.lia.ufc.br/es10.1/arquivos/CAP03-resumo.pdf⟩.

ARDUPILOT MEGA. *ArduPilot*. Disponível em: ⟨https://www.ardupilot.co.uk/⟩. Acesso em: nov. 2018.

BRUN, M.; DELATOUR, J.; TRINQUET, Y. Code generation from aadl to a real-time operating system: An experimentation feedback on the use of model transformation. In: IEEE. *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*. [S.l.], 2008. p. 257–262.

ECLIPSE FOUNDATION. *Eclipse Process Framework Composer*. 2018. Disponível em: ⟨https://projects.eclipse.org/projects/technology.epf⟩. Acesso em: nov. 2018.

FEILER, P. H.; GLUCH, D. P.; HUDAK, J. J. *The architecture analysis & design language (AADL): An introduction*. [S.l.], 2006.

FIGUEIREDO, E. processo de desenvolvimento dirigido por modelos (mdd). In: . [S.l.]: DCC/ICEx/UFMG, 2011.

GUI, S. et al. Ucag: an automatic c code generator for aadl based upon deltaos. In: IEEE. *2008 International Conference on Advanced Computer Theory and Engineering*. [S.l.], 2008. p. 346–350.

HOLLOWAY, C. M. Towards understanding the do-178c/ed-12c assurance case. In: IET. *System Safety, incorporating the Cyber Security Conference 2012, 7th IET International Conference on*. [S.l.], 2012. p. 1–6.

MA, L. et al. Research of automatic code generating technology based on aadl. In: IEEE. *Embedded Software and Systems Symposia, 2008. ICESS Symposia'08. International Conference on*. [S.l.], 2008. p. 136–141.

MISSIONPLANNER. *ArduPilot*. Disponível em: ⟨https://github.com/ArduPilot/MissionPlanner⟩.

OBJECT MANAGEMENT GROUP. *SPEM 2.0*. 2008. Disponível em: ⟨https://www.omg.org/spec/SPEM/About-SPEM/⟩. Acesso em: nov. 2018.

OLIVEIRA, A. L. de. *Variability management in safety-critical software product line engineering*. [S.l.], 2018.

OLIVEIRA, R. G. d. et al. Contribuições para melhoria do processo de verificação formal de propriedades em programas aadl. 2011.

PALMA, A. E. T. P. da; QUINÁIA, M. A. Reengenharia de software: o que, por quê e como. *RECEN-Revista Ciências Exatas e Naturais*, v. 1, n. 2, p. 33–51, 2010.

PASINI, K. et al. Uma solução para apoiar um processo de desenvolvimento dirigido a modelos usando openarchitectureware. In: *IX Free Software Workshop/9th Intl. Forum of Free Software*. [S.l.: s.n.], 2008. p. 121–126.

SELIC, B. The pragmatics of model-driven development. *IEEE software*, IEEE, v. 20, n. 5, p. 19–25, 2003.

SINHA, P. Architectural design and reliability analysis of a fail-operational brake-by-wire system from iso 26262 perspectives. *Reliability Engineering & System Safety*, Elsevier, v. 96, n. 10, p. 1349–1359, 2011.

SOMMERVILLE, I. *Software Engineering (6th Ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-39815-X.

A Código Exemplo ArduPilot

Listing A.1: Código exemplo do ArduPilot parete 1

```
1
2  using System;
3  using System.Windows.Forms;
4  using MissionPlanner.Mavlink;
5  using MissionPlanner.Utilities;
6
7  namespace MissionPlanner.Controls{
8
9      public partial class AuthKeys : Form{
10
11         public AuthKeys(){
12
13             InitializeComponent();
14             ThemeManager.ApplyThemeTo(this);
15             LoadKeys();
16
17         }
18
19         private void but_save_Click(object sender, EventArgs e){
20
21             Save();
22
23         }
24
25         private void LoadKeys(){
26
27             dataGridView1.Rows.Clear();
28
29             foreach (var authKey in MAVAuthKeys.Keys){
30
31                 int row = dataGridView1.Rows.Add();
32                 dataGridView1[FName.Index, row].Value = authKey.
                       Key;
33                 dataGridView1[Key.Index, row].Value = Convert.
                       ToBase64String(authKey.Value.Key);
34
35             }
36         }
37
38         private void Save(){
39
40             MAVAuthKeys.Save();
41
42         }
```

Listing A.2: Código exemplo do ArduPilot parte 2

```
1
2         private void dataGridView1_CellContentClick(object sender
              , DataGridViewCellEventArgs e)
3         {
4
5             if (e.ColumnIndex == Use.Index)
6             {
7
8                 MainV2.comPort.setupSigning("",Convert.
                      FromBase64String(dataGridView1[Key.Index,e.
                      RowIndex].Value.ToString()));
9
10            }
11        }
12
13        private void but_add_Click(object sender, EventArgs e)
14        {
15
16            int row = dataGridView1.Rows.Add();
17
18            string name = "";
19
20            if (InputBox.Show("Name", "Please enter a friendly
                  name", ref name) == DialogResult.OK)
21            {
22
23                dataGridView1[FName.Index, row].Value = name;
24
25                string pass = "";
26
27                if (InputBox.Show("Input Seed", "Please enter
                      your pass prase", ref pass) == DialogResult.OK)
28                {
29
30                    var input = InputBox.value;
31
32                    MAVAuthKeys.AddKey(dataGridView1[FName.Index,
                          row].Value.ToString(), input);
33
34                }
35
36                dataGridView1.EndEdit();
37
38                Save();
39
40                LoadKeys();
41
42            }
43        }
```

Listing A.3: Código exemplo do ArduPilot parte 3

```
1
2         private void dataGridView1_UserDeletedRow(object sender,
              DataGridViewRowEventArgs e)
3         {
4             MAVAuthKeys.Keys.Remove(e.Row.Cells[FName.Index].
                  Value.ToString());
5         }
6
7         private void dataGridView1_RowsAdded(object sender,
              DataGridViewRowsAddedEventArgs e)
8         {
9             dataGridView1[Use.Index, e.RowIndex].Value = "Use";
10        }
11
12        private void but_disablesigning_Click(object sender,
              EventArgs e)
13        {
14            MainV2.comPort.setupSigning("");
15        }
16    }
17}
```

Listing A.4: Código exemplo do ArduPilot parte 1

```
1 using System;
2 using System.Collections.Generic;
3 using System.Drawing;
4 using System.Drawing.Imaging;
5 using System.IO;
6 using System.Runtime.InteropServices;
7
8 namespace MissionPlanner.Utilities
9 {
10    public class OpticalFlow: IDisposable
11    {
12        // since its not listed in the docs
13        // https://github.com/PX4/Flow/blob/4
              a314cfdb099aed9795b825e7518203771207fbb/src/modules/
              flow/dcmi.c#L292
14
15        // size of incomming data
16        MAVLink.mavlink_data_transmission_handshake_t
              msgDataTransmissionHandshake;
17
18        // data from handshake
19        MAVLink.mavlink_encapsulated_data_t msgEncapsulatedData;
20
21        private KeyValuePair<MAVLink.MAVLINK_MSG_ID, Func<MAVLink
              .MAVLinkMessage, bool>> subDataTrans;
22        private KeyValuePair<MAVLink.MAVLINK_MSG_ID, Func<MAVLink
              .MAVLinkMessage, bool>> subEncapData;
```

Listing A.5: Código exemplo do ArduPilot parte 2

```
1
2         MAVLinkInterface _mav;
3
4         MemoryStream imageBuffer = new MemoryStream();
5
6         public event EventHandler<ImageEventHandle> newImage;
7
8         public class ImageEventHandle : EventArgs
9         {
10            public Image Image { get; set; }
11
12            public ImageEventHandle(Image bmp)
13            {
14                Image = bmp;
15            }
16        }
17
18        public OpticalFlow(MAVLinkInterface mav)
19        {
20            _mav = mav;
21
22            subDataTrans = mav.SubscribeToPacketType(MAVLink.
                  MAVLINK_MSG_ID.DATA_TRANSMISSION_HANDSHAKE,
                  ReceviedPacket);
23            subEncapData = mav.SubscribeToPacketType(MAVLink.
                  MAVLINK_MSG_ID.ENCAPSULATED_DATA, ReceviedPacket);
24        }
25
26        public void CalibrationMode(bool on_off = false)
27        {
28            if (on_off == true)
29                _mav.setParam("VIDEO_ONLY", 1);
30            else
31                _mav.setParam("VIDEO_ONLY", 0);
32        }
33
34        private bool ReceviedPacket(MAVLink.MAVLinkMessage arg)
35        {
36            if (arg.msgid == (byte) MAVLink.MAVLINK_MSG_ID.
                  DATA_TRANSMISSION_HANDSHAKE)
37            {
38
39                var packet = arg.ToStructure<MAVLink.
                      mavlink_data_transmission_handshake_t>();
40                msgDataTransmissionHandshake = packet;
41                imageBuffer.Close();
42                imageBuffer = new MemoryStream((int)packet.size);
43
44            }
```

Listing A.6: Código exemplo do ArduPilot parte 3

```
1
2             else if (arg.msgid == (byte)MAVLink.MAVLINK_MSG_ID.
                  ENCAPSULATED_DATA)
3             {
4                 var packet = arg.ToStructure<MAVLink.
                      mavlink_encapsulated_data_t>();
5                 msgEncapsulatedData = packet;
6                 int start = msgDataTransmissionHandshake.payload
                      * msgEncapsulatedData.seqnr;
7                 int left = (int)msgDataTransmissionHandshake.size
                      - start;
8                 var writeamount = Math.Min(msgEncapsulatedData.
                      data.Length, left);
9
10                imageBuffer.Seek(start, SeekOrigin.Begin);
11                imageBuffer.Write(msgEncapsulatedData.data, 0,
                      writeamount);
12
13                // we have a complete image
14                if ((msgEncapsulatedData.seqnr+1) ==
                      msgDataTransmissionHandshake.packets)
15                {
16                    using (
17                        Bitmap bmp = new Bitmap(
                              msgDataTransmissionHandshake.width,
                              msgDataTransmissionHandshake.height,
18                                PixelFormat.Format8bppIndexed))
19                    {
20                        SetGrayscalePalette(bmp);
21
22                        var bitmapData = bmp.LockBits(new
                              Rectangle(Point.Empty, bmp.Size),
                              ImageLockMode.ReadWrite,
23                                bmp.PixelFormat);
24
25                        if (imageBuffer.Length >
                              msgDataTransmissionHandshake.size)
26                            return true;
27                        var buffer = imageBuffer.ToArray();
28                        Marshal.Copy(buffer, 0, bitmapData.Scan0,
                              buffer.Length);
29                        bmp.UnlockBits(bitmapData);
30                        if (newImage != null)
31                            newImage(this, new ImageEventHandle(
                                  bmp));
32                        //bmp.Save("test.bmp", ImageFormat.Bmp);
33                    }
34                }
35            }
36            return true;
37        }
```

Listing A.7: Código exemplo do ArduPilot parte 4

```
1
2           public void SetGrayscalePalette(Bitmap bmp)
3           {
4               ColorPalette _palette = bmp.Palette;
5               for (int i = 0; i < 256; i++)
6                   _palette.Entries[i] = Color.FromArgb(255, i, i, i
                        );
7               bmp.Palette = _palette;
8           }
9
10          public void Close()
11          {
12              _mav.UnSubscribeToPacketType(subDataTrans);
13              _mav.UnSubscribeToPacketType(subEncapData);
14
15              imageBuffer.Close();
16          }
17
18          public void Dispose()
19          {
20              Close();
21          }
22      }
23  }
```

Listing A.8: Código exemplo do ArduPilot em AADL parte 1

```
1   package Controls
2
3   public
4
5       with Mavlink;
6       with Utilities;
7       with Base_Types;
8
9       data object
10      end object;
11
12      data EventArgs
13      end EventArgs;
14
15      data DataGridViewRowEventArgs
16      end DataGridViewRowEventArgs;
17
18      data DataGridViewRowsAddedEventArgs
19      end DataGridViewRowsAddedEventArgs;
20
21      process AuthKeys extends Form -- classe do sistema System.
            Windows.Forms
22      end AuthKeys;
```

Listing A.9: Código exemplo do ArduPilot em AADL parte 2

```
1
2       process implementation AuthKeys.imp
3            subcomponents
4                 Contrutor: thread AuthKeysC;
5                 IC: subprogram InitializeComponent;
6                 LK: subprogram LoadKeys;
7            connections
8                 C0: port Key -> Contrutor.Key;
9       end AuthKeys.imp;
10
11      process Form
12           features
13                Key: in out data port Base_Types::Integer;
14      end Form;
15
16      process implementation Form.imp
17      end Form.imp;
18
19      thread AuthKeysC
20           features
21                Key: in out data port Base_Types::Integer;
22      end AuthKeysC;
23
24      thread implementation AuthKeysC.imp
25           calls MyCall: {
26                IC: subprogram InitializeComponent;
27                LK: subprogram LoadKeys;
28           };
29           connections
30                C0: parameter key -> LK.Key;
31      end AuthKeysC.imp;
32
33      subprogram InitializeComponent
34      end InitializeComponent;
35
36      subprogram implementation InitializeComponent.imp
37      end InitializeComponent.imp;
38
39      subprogram LoadKeys
40           features
41                row: in out parameter Base_Types::Integer;
42                Key: in parameter Base_Types::Integer;
43      end LoadKeys;
44
45      subprogram implementation LoadKeys.imp
46           calls MyCall:{
47                dGW: subprogram dataGridView1;
48           };
49           connections
50                C0: parameter Key -> dGW.Value;
51      end LoadKeys.imp;
```

Listing A.10: Código exemplo do ArduPilot em AADL parte 3

```
1       subprogram dataGridView1
2            features
3                 Value: in parameter Base_Types::Integer;
4                 Values: in parameter Base_Types::String;
5       end dataGridView1;
6
7       subprogram implementation dataGridView1.imp
8            calls MyCall:{
9                 Rows: subprogram Rows;
10                EndEdit: subprogram EndEdit;       };
11      end dataGridView1.imp;
12
13      subprogram Rows
14      end Rows;
15
16      subprogram implementation Rows.imp
17           calls MyCall:{
18                Clear: subprogram Clear;
19                Add: subprogram Add;};
20      end Rows.imp;
21
22      subprogram Clear
23      end Clear;
24
25      subprogram implementation Clear.imp
26      end Clear.imp;
27
28      subprogram Add
29      end Add;
30
31      subprogram implementation Add.imp
32      end Add.imp;
33
34      subprogram EndEdit
35      end EndEdit;
36
37      subprogram implementation EndEdit.imp
38      end EndEdit.imp;
39 private
40      with Controls;
41
42      thread but_save_Click
43      end but_save_Click;
44
45      thread implementation but_save_Click.imp
46           subcomponents
47                sender: data object;
48                e: data EventArgs;
49           calls MyCall: {
50                S: subprogram save; };
51      end but_save_Click.imp;
```

Listing A.11: Código exemplo do ArduPilot em AADL parte 4

```
1
2       subprogram save
3       end save;
4
5       subprogram implementation save.imp
6            --calls MyCall:{
7                 --Save: subprogram MAVAuthKeys::Save;--Chamada de
                     fun   o do Form--};
8       end save.imp;
9
10      thread dataGridView1_CellContentClick
11           features
12                sender: in data port object;
13                e: in data port  DataGridViewRowEventArgs;
14      end dataGridView1_CellContentClick;
15
16      thread implementation dataGridView1_CellContentClick.imp
17      end dataGridView1_CellContentClick.imp;
18
19      thread but_add_Click
20           features
21                sender: in data port object;
22                e: in data port EventArgs;
23      end but_add_Click;
24
25      thread implementation but_add_Click.imp
26           subcomponents
27                row: data Base_Types::Integer;
28                name: data Base_Types::String;
29                pass: data Base_Types::String;
30                input: data Base_Types::Unsigned_32;
31           calls MyCall:{
32                dGV: subprogram dataGridView1;
33                Save: subprogram save;
34                LK: subprogram LoadKeys;
35                --AddKey: subprogram MAVAuthKeys::AddKey;};
36           connections
37                C0: parameter name -> dGV.Values;
38      end but_add_Click.imp;
39
40      thread dataGridView1_UserDeletedRow
41           features
42                sender: in data port object;
43                e: in data port DataGridViewRowEventArgs;
44      end dataGridView1_UserDeletedRow;
45
46      thread implementation dataGridView1_UserDeletedRow.imp
47           --calls MyCall:{
48                --Remove: subprogram MAVAuthKeys::Remove;--Chamada
                     de fun   o do Form;--};
49      end dataGridView1_UserDeletedRow.imp;
```

Listing A.12: Código exemplo do ArduPilot em AADL parte 5

```
1
2       thread dataGridView1_RowsAdded
3            features
4                 sender: in data port object;
5                 e: in data port DataGridViewRowsAddedEventArgs;
6       end dataGridView1_RowsAdded;
7
8       thread implementation dataGridView1_RowsAdded.imp
9            subcomponents
10                Use: data Base_Types::String;
11           calls MyCall:{
12                dGV: subprogram dataGridView1;
13           };
14           connections
15                C0: parameter Use -> dGV.Values;
16      end dataGridView1_RowsAdded.imp;
17
18      thread but_disablesigning_Click
19           features
20                sender: in data port object;
21                e: in data port EventArgs;
22      end but_disablesigning_Click;
23
24      thread implementation but_disablesigning_Click.imp
25           --calls MyCall:{
26                --comPort: subprogram MainV2::comPort;--Chamada de
                       fun   o do Form
27           --};
28      end but_disablesigning_Click.imp;
29
30 end Controls;
```

Listing A.13: Código exemplo do ArduPilot em AADL parte 1

```
1  package Utilities
2  public
3       with Base_Types;
4       data KeyValuePair
5       end KeyValuePair;
6
7       data MAVLinkInterface
8       end MAVLinkInterface;
9
10      data MemoryStream
11      end MemoryStream;
12
13      data EventHandler
14      end EventHandler;
15
16      data Image
17      end Image;
```

Listing A.14: Código exemplo do ArduPilot em AADL parte 2

```
1      data MAVLinkMessage
2      end MAVLinkMessage;
3      data Bitmap
4      end Bitmap;
5
6      data ColorPalette
7      end ColorPalette;
8
9      process OpticalFlow extends IDisposable
10          features
11              subDataTrans: in out data port KeyValuePair;
12              subEncapData: in out data port KeyValuePair;
13              mav: in out data port MAVLinkInterface;
14              imageBuffer: in out data port MemoryStream;
15              newImage: in out event data port EventHandler;
16     end OpticalFlow;
17
18     process implementation OpticalFlow.imp
19          subcomponents
20              OFC: thread OpticalFlowC;
21              CM: thread CalibrationMode;
22          connections
23              C0: port OFC.mav -> mav;
24     end OpticalFlow.imp;
25
26     process IDisposable
27     end IDisposable;
28
29     process implementation IDisposable.imp
30     end IDisposable.imp;
31
32     process ImageEventHandle
33          features
34              Image: in out data port Image;
35     end ImageEventHandle;
36
37     process implementation ImageEventHandle.imp
38          subcomponents
39              IH: thread ImageEventHandleC;
40          connections
41              C0: port IH.bmp -> Image;
42     end ImageEventHandle.imp;
43
44     thread ImageEventHandleC
45          features
46              bmp: in out data port Image;
47     end ImageEventHandleC;
48
49     thread implementation ImageEventHandleC.imp
50     end ImageEventHandleC.imp;
```

Listing A.15: Código exemplo do ArduPilot em AADL parte 3

```
1
2       thread OpticalFlowC
3            features
4                 mav: in out data port MAVLinkInterface;
5       end OpticalFlowC;
6
7       thread implementation OpticalFlowC.imp
8            subcomponents
9                 subDataTrans: data MAVLinkInterface;
10                subEncapData: data MAVLinkInterface;
11      end OpticalFlowC.imp;
12
13      thread CalibrationMode
14           features
15                on_off: in data port Base_Types::Boolean;
16      end CalibrationMode;
17
18      thread implementation CalibrationMode.imp
19      end CalibrationMode.imp;
20
21      thread ReceviedPacket
22           features
23                bol: out data port Base_Types::Boolean;
24                arg: in out data port MAVLinkMessage;
25      end ReceviedPacket;
26
27      thread implementation ReceviedPacket.imp
28           subcomponents
29                packet: data Base_Types::Unsigned_32;
30                start: data Base_Types::Integer;
31                left: data Base_Types::Integer;
32                writeamount: data Base_Types::Unsigned_32;
33                bmp: data Bitmap;
34                bitmapData: data Base_Types::Unsigned_32;
35                buffer: data Base_Types::Unsigned_32;
36           calls MyCall:{
37                SGP: subprogram SetGrayscalePalette;
38           };
39           connections
40                C0: parameter bmp -> SGP.bmp;
41      end ReceviedPacket.imp;
42
43      subprogram SetGrayscalePalette
44           features
45                bmp: in parameter Bitmap;
46                palette: in out parameter ColorPalette;
47      end SetGrayscalePalette;
48
49      subprogram implementation SetGrayscalePalette.imp
50      end SetGrayscalePalette.imp;
```

Listing A.16: Código exemplo do ArduPilot em AADL parte 1

```
     subprogram Close
     end Close;

     subprogram implementation Close.imp
          calls MyCall:{
               IB: subprogram imageBuffer;
          };
     end Close.imp;

     subprogram imageBuffer
     end imageBuffer;

     subprogram implementation imageBuffer.imp
     end imageBuffer.imp;

     thread Dispose
     end Dispose;

     thread implementation Dispose.imp
          calls MyCall:{
               C: subprogram Close;
          };
     end Dispose.imp;

end Utilities;
```