UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Rosane Fátima Passarini

TRANSFORMAÇÃO ASSISTIDA DE MODELOS: MECANISMO DE SUPORTE PARA O DESENVOLVIMENTO DE *CYBER-PHYSICAL SYSTEMS*

Florianópolis

2014

Rosane Fátima Passarini

TRANSFORMAÇÃO ASSISTIDA DE MODELOS: MECANISMO DE SUPORTE PARA O DESENVOLVIMENTO DE *CYBER-PHYSICAL SYSTEMS*

Tese submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas para a obtenção do Grau de Doutor em Engenharia de Automação e Sistemas.
Orientador: Prof. Leandro Buss Becker, Dr.
Coorientador: Prof. Jean-Marie Farines, Dr.

Florianópolis

2014

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Passarini, Rosane Fátima
Transformação Assistida de Modelos: Mecanismo de suporte para o desenvolvimento de Cyber Physical Systems / Rosane Fátima Passarini; orientador, Leandro Buss Becker; coorientador, Jean-Marie Farines. - Florianópolis, SC, 2014.
170 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de Automação e Sistemas. 2. Engenharia Dirigida por Modelos. 3. Transformação de Modelos. 4. Modelagem Funcional. 5. Modelagem da Arquitetura do Sistema. I. Becker, Leandro Buss. II. Farines, Jean-Marie. III. Universidade Federal de Santa Catarina. Programa de ós-Graduação em Engenharia de Automação e Sistemas. IV.Título.

Rosane Fátima Passarini

TRANSFORMAÇÃO ASSISTIDA DE MODELOS: MECANISMO DE SUPORTE PARA O DESENVOLVIMENTO DE *CYBER-PHYSICAL SYSTEMS*

Esta Tese foi julgada aprovada para a obtenção do Título de "Doutor em Engenharia de Automação e Sistemas", e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Florianópolis, 29 de agosto 2014.

Prof. Leandro Buss Becker, Dr.
Orientador

Prof. Jean-Marie Farines, Dr.
Coorientador

Prof. Chefe, Rômulo Silva de Oliveira, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Leandro Buss Becker, Dr.
Presidente

Prof. Raimundo Barreto, Dr. - UFMA

Prof. Marco Aurélio Wehrmeister, Dr. - UTFPR/Curitiba

Prof. Cristian Koliver, Dr. - IFC/Camboriu

Prof. Max Hering de Queiroz, Dr. - UFSC/Florianópolis

Prof. Jomi Fred Hubner, Dr. - UFSC/Florianópolis

Dedico este trabalho a minha família, pelo apoio, carinho e compreensão.

AGRADECIMENTOS

Agradeço à Deus e a meu paizinho que sempre estiveram comigo iluminando e guiando meu caminho.

Manifesto também meus sinceros agradecimentos aos professores Leandro Buss Becker e Jean-Marie Farines pela confiança depositada em minha pessoa, pela valiosa orientação, paciência e apoio durante a realização deste trabalho de pesquisa.

Agradeço também aos meus colegas da Universidade Tecnológica Federal do Paraná câmpus Toledo, por me proporcionarem condições para o desenvolvimento deste trabalho.

"Somos todos geniais. Mas se julgar um peixe por sua capacidade de subir em árvores, ele passará sua vida inteira acreditando ser um estúpido"

Albert Einstein

RESUMO

O termo *Cyber-Physical System* representa um dispositivo eletromecânico controlado por um sistema baseado em computador, exemplos deste tipo de sistema incluem robôs, aviões, redes inteligentes, entre outros. Devido à natureza multidisciplinar dos *Cyber-Physical Systems*, eles normalmente são projetados utilizando diferentes modelos. A perspectiva "cibernétca" deste tipo de sistema pressupõe a existência de: (i) um modelo matemático que representa a dinâmica do sistema físico, (ii) algoritmos de controle, e (iii) um projeto do sistema computacional embarcado. Dentro deste contexto, esta tese de doutorado investiga uma forma de abordar adequadamente o projeto do sistema computacional embarcado de um *Cyber-Physical System* baseada na modelagem funcional do mesmo. Buscando evitar desta forma a criação de modelos funcionais e arquitetônicos dissociados, e além disso, promover uma abordagem de projeto dirigido por modelos, proporcionando benefícios como a independência de plataforma, níveis de abstração mais altos, e a reutilização de informações. Como resultado da pesquisa realizada, é apresentada uma solução que ajuda a realizar a transição do modelo funcional para o modelo de arquitetura de software durante o processo de desenvolvimento de um *Cyber-Physical System*. Para isso, é sugerido como relacionar elementos de um modelo funcional com elementos de um modelo de arquitetura. A solução proposta, chamada de "Transformação Assistida de Modelos (AST)", fornece suporte para a transformação de modelos Simulink utilizados para a modelagem funcional em modelos arquitetônicos expressos em AADL, e aumenta a confiabilidade de que os modelos funcional e arquitetural são consistentes entre si, uma vez que diminui ocorrência de erros de inconsistência de interface (portas, tipos de dados e conexões) entre os mesmos. A AST contribui portanto, com a implantação/integração de aplicativos verificados em arquiteturas validadas tornando o processo de desenvolvimento de *Cyber-Physical Systems* mais robusto. Durante os experimentos, realizados na forma de estudos de caso, os modelos gerados pela AST mostraram-se passíveis de análises sintáticas, verificações comportamentais, e análises de escalonabilidade e de latência de fluxos, o que serviu para reforçar a escolha pelo de o uso de modelos AADL durante o processo de desenvolvimento de CPS. Também foi implementado no escopo desta pesquisa, o protótipo de uma ferramenta computacional que automatiza a aplicação da solução

proposta. O protótipo foi implementado utilizando a linguagem de programação Java, e empacotado como um plugin para ser usado dentro do ambiente OSATE (*Open Source Architectural Environment Tool*), que é um processador de modelos AADL que roda dentro do Eclipse. O plugin em questão, chamado de AS2T, também pode ser considerado uma alternativa para estender a cadeia de transformação de modelos do ambiente TOPCASED, que é um ambiente *OpenSource* para desenvolvimento de sistemas embarcados críticos que também faz uso do OSATE.
Palavras-chave: MDE. Transformação de Modelos. Modelo Funcional. Arquitetura de Software. AADL. Simulink

ABSTRACT

Cyber-Physical System (CPS) is a denomination used to represent an electro-mechanical device controlled by a computerized system. Examples of CPS include robots, airplanes, smart grids, among others. Due to the multidisciplinary nature of CPSs, they are normally designed using different models. The "cybernetic" perspective assumes the existence of: (i) a mathematical model that represents the dynamics of the physical system, (ii) some control algorithms, and (iii) a design of the embedded computing system. In this context, this thesis investigates a way to adequately address the design of the architecture embedded computing system of a CPS based on a preliminary functional model. Looking forward to avoid the creation of decoupled functional and architectural models and aiming to promote a model-based design approach for CPS, the proposed approach targets using higher levels of abstraction and model-information reuse. The solution presented in this thesis is named "Assisted Transformation of Models" (AST), it focuses on discussing how to related elements of a functional model with the elements of an architectural model. AST provides support for the transformation of the Simulink models used for the functional modeling into architectural models expressed in AADL. As benefits of using the proposed solution, one can see that it increases the reliability that the functional and architectonical models are consistent between themselves, especially when considering the connection interfaces between components (ports and connections data types). Experiments were conducted to validate the proposed transformation process. The generated models were analyzed in respect to the syntax correctness and also regarding additional model analyses, such as behavioral verification and schedulability analysis. The work provides a prototype tool that automates the proposed transformation process. Such tool can be used as plugin from OSATE (Open Source Architectural Environment Tool), which is an AADL processor that runs within Eclipse. The AS2T plugin can be considered an alternative to extend the chain of transformation of models of the TOPCASED environment, which is an OpenSource development environment of critical embedded systems that makes use of OSATE.

Keywords: MDE. Models Transformation. Functional Model. Software Architecture. AADL. Simulink

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUCÃO

Ao longo deste capítulo são apresentadas as bases que fundamentam este trabalho de pesquisa. Primeiramente o problema de pesquisa é contextualizado e justificado. Na sequência apresenta-se o objetivo geral e os objetivos específicos. Depois é apresentada a forma como a pesquisa foi conduzida e uma síntese do conteúdo dos capítulos que compõem este documento de tese.

1.1 APRESENTAÇÃO

Em vários domínios de aplicação os sistemas embarcados são uma parte essencial de produtos, sistemas e soluções. Um sistema embarcado pode ser definido como sendo um sistema de computação, cujo o processamento da informação está integrado a um dispositivo físico ou a um sistema que ele controla, com o objetivo de estender e otimizar suas funcionalidades (MARWEDEL, 2011), (JENSEN; CHANG; LEE, 2011).

Esses sistemas empregam recursos da ciência e da engenharia da computação para melhorar e ampliar as funcionalidades de sistemas mecânicos, elétricos, térmicos, químicos, biológicos e suas combinações (KIM, 2010). Quando há um forte foco na interação entre diferentes dispositivos físicos, o sistema resultante está sendo qualificado como um *Cyber-Physical System* (CPS).

Cyber-Physical System (CPS) é o termo que está sendo adotado atualmente para qualificar sistemas computacionais embarcados que encontram-se intimamente ligados a dispositivos físicos (LEE, 2008). É possível considerar que os *Cyber-Physical Systems* referem-se a uma nova geração de sistemas embarcados, a qual explora as capacidades de interação e expansão das funcionalidades dos dispositivos do mundo físico através da computação, comunicação e controle (BAHETI; GILL, 2011). Tal cenário está representado graficamente na Figura 1

De forma geral, é possível considerar que os CPSs tem um papel a desempenhar no desenvolvimento de uma nova teoria de sistemas físicos mediados por computador (BAHETI; GILL, 2009). Uma vez que, desafios consideráveis são encontrados durante o processo de desenvolvimento deste tipo de sistema, principalmente, porque os componentes físicos destes sistemas introduzem requisitos de segurança e confiabilidade qualitativamente diferentes daqueles comumente encontrados na com-

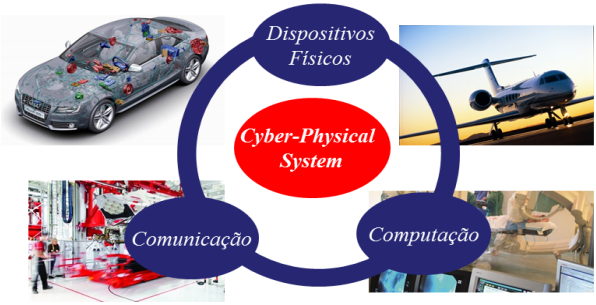

Figura 1 – Representação Gráfica de CPSs

putação de propósito geral. Além disso, a grande quantidade de funcionalidades que está sendo associada a este tipo de sistema também contribui com o aumento da complexidade dos mesmos.

A tendência de aumento do uso dos CPSs oferece novas oportunidades, tanto no espaço de pesquisa, como para novas aplicações. Oportunidades e desafios de pesquisa nesta área incluem o projeto e o desenvolvimento da nova geração de aeronaves, veículos espaciais, veículos automotores, robótica móvel, condução urbana totalmente autônoma, próteses que permitem que os sinais do cérebro controlem objetos físicos, entre outros. A linha de pesquisa que explora a integração virtual dos diferentes tipos de modelos utilizados durante o processo de desenvolvimento de um sistema embarcado possui uma comunidade bastante ativa atualmente.

O desenvolvimento orientado a modelo (MDD) é um paradigma de engenharia que facilita a definição, composição e integração de sistemas de software complexos. Nele a evolução de um modelo ocorre através de refinamentos, transformações e geração de código, possivelmente automatizado com o apoio de ferramentas, formando a base da engenharia dirigida a modelos (MDE). Na visão da MDE, modelos de software são elevados a um papel central e regem o processo de desenvolvimento, atingindo um nível mais alto de abstração do que é possível com as linguagens atuais de programação de terceira geração (SELIC, 2007), (MAZZINI; PURI; VARDANEGA, 2009).

Pesquisas sobre projeto de sistemas embarcados apontam que o desenvolvimento a partir de níveis mais elevados de abstração podem contribuir com o sucesso de um projeto. Alguns autores, como (SELIC, 2003) e (SCHMIDT, 2006) argumentam há algum tempo que a MDE é uma forma viável de lidar com a complexidade encontrada nas

novas gerações de sistemas embarcados. Usando essa abordagem, os modelos de sistemas embarcados devem evoluir a partir de visões de alto nível para implementações reais, garantindo um processo relativamente suave e potencialmente mais confiável em comparação com as formas tradicionais de engenharia.

O desenvolvimento de sistemas embarcados também envolve a integração de vários domínios, sendo que a cooperação destes domínios é essencial para o êxito do desenvolvimento destes sistemas. Neste sentido, várias ferramentas de modelagem surgiram ou se adaptaram ao conceito da engenharia dirigida por modelos e são capazes de gerar código, facilitando consideravelmente o processo de desenvolvimento de sistemas embarcados. No entanto, estas ferramentas ou se concentram nos aspectos funcionais da aplicação, ou em aspectos da arquitetura em tempo de execução, dividindo assim, o desenvolvimento em partes com notações de modelagem heterogêneas e com uma coordenação fraca (DELANGE et al., 2010).

1.2 CONTEXTUALIZAÇÃO DO PROBLEMA DE PESQUISA

O fato de um sistema computacional precisar controlar um sistema físico (por completo ou apenas uma parte) faz com que o processo de desenvolvimento de um CPS seja uma atividade multidisciplinar. O projeto completo de um CPS normalmente é composto pelo projeto da dinâmica do sistema físico, pelo projeto dos algoritmos de controle e pelo projeto do sistema computacional embarcado (SCE) (LEE, 2008).

O projeto da dinâmica do sistema físico representa a criação de uma modelagem matemática do dispositivo eletromecânico a ser controlado. O que pode ser tão simples como uma variável booleana, o que poderia, por exemplo, representar o estado de um LED, ou tão complexo como um conjunto de equações diferenciais, o que poderia, por exemplo, representar a dinâmica de um Veículo Aéreo Não Tripulado (UAV). Uma das razões para a modelagem da dinâmica do sistema físico é a possibilidade de testar os algoritmos de controle que são desenvolvidos ao longo do projeto.

O Simulink é o exemplo de uma ferramenta usualmente adotada pelos engenheiros de controle para a modelagem e simulação dos CPSs, o que abrange a concepção da dinâmica do sistema físico e os algoritmos de controle. Embora, esta ferramenta ofereça a possibilidade de geração de código, ela tem um fraco suporte para a expressão e geração do projeto do SCE (GONCALVES et al., 2013). Ao considerar o projeto

de um CPS, genericamente, as mesmas características e desvantagens se aplicam a outras ferramentas de simulação relacionadas, como o LabVIEW, Modelica, Scilab e outros. Assim, cabe ao engenheiro de computação reprojetar a arquitetura do SCE a partir do código gerado pela ferramenta de modelagem e simulação usada. Além de ineficiente, esta prática causa um desacoplamento entre o modelo funcional e o modelo arquitetural do sistema.

Linguagens de descrição de arquitetura, em particular a Linguagem de Projeto e Análise de Arquitetura (AADL), são adequadas para representar o projeto do SCE. Além de permitir expressar em detalhes a organização do software, a AADL também fornece meios para expressar a plataforma de hardware e também a sua associação com os componentes de software. Além disso, ela possui suporte de ferramentas adequadas para realizar diversos tipos de análises e verificações do modelo, o que permite reduzir o risco de erros de projeto (P. Feiler and J. Hudak., 2006).

Como pode ser observado na Figura 16, os modelos funcionais e de arquitetura gerados durante o processo de desenvolvimento de um CPS podem ser considerados complementares, uma vez que os modelos funcionais fornecem o comportamento (geralmente na forma de código fonte) para os componentes de software do modelo de arquitetura.

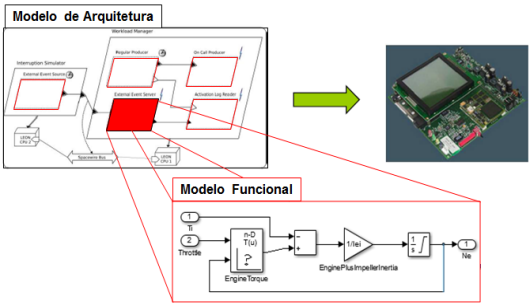

Figura 2 – Relação entre Modelos Funcionais e o Modelo de Arquitetura de um CPS

De acordo com Raghav et.al (RAGHAV et al., 2009b), o frequente desacoplamento entre os modelos funcionais e o modelo de arquitetura do sistema acontece porque estes modelos fornecem diferentes abstrações do sistema que está sendo projetado. Ou seja, os modelos funcionais contém informações que expressam as funcionalidades de um CPS. Por outro lado, o modelo de arquitetura do sistema (modelo de

arquitetura de software + modelo de arquitetura de hardware), que compõem o projeto do SCE, se preocupa com o número de processadores, com a organização do software em termos de processos e threads, e em como eles são escalonados.

Neste cenário, percebe-se que uma dificuldade eminente é a forma de como se deve estabelecer a ligação entre os componentes do modelo funcional (técnicas de controle) e os componentes do modelo de arquitetura de software do SCE. Isso acontece porque que os modelos funcionais e de arquitetura contêm abstrações distintas do sistema em desenvolvimento. Assim, considerando (i) que se faz necessário à utilização de diferentes tipos de modelos para se projetar um CPS e (ii) que estes modelos precisam ser consistentes entre si, esta tese de doutorado apresenta um mecanismo que auxilia a transição entre a modelagem funcional e a modelagem arquitetural durante o processo de desenvolvimento de CPSs, a qual é considerada por muitos uma transição de projeto bastante crítica.

O mecanismo apresentado neste documento de tese, chamado de Transformação Assistida de Modelos (AST), oferece suporte para a extração de informações de um modelo funcional, modelo este, gerado em uma etapa anterior do processo de desenvolvimento, sem que para isso seja necessário dominar técnicas de controle, nem tão pouco analisar o código fonte gerado pelas ferramentas de modelagem e simulação. Para tanto, foram empregadas técnicas da MDE para definir um conjunto de regras de transformação capaz de executar a transformação de modelos funcionais, utilizados para a modelagem funcional e simulações, em modelos de arquitetura.

Na AST um modelo funcional, que foi gerado levando em consideração um conjunto de diretrizes de modelagem, é importado e um motor de transformação executa um conjunto de transformações previamente definido, gerando como resultado um modelo preliminar do modelo de arquitetura de software do SCE, modelo este, hierarquicamente estruturado conforme o modelo funcional importado. No contexto da AST, os modelos funcionais devem representar a topologia do sistema, a qual descreve a hierarquia estrutural do sistema de controle por meio de subsistemas, suas interfaces e conexões. Os subsistemas representam as funcionalidades do CPS extraídas da especificação de requisitos funcionais do sistema.

O mecanismo proposto nesta tese de doutorado encontra-se situado no escopo de um projeto mais abrangente, o qual busca prover um processo de desenvolvimento baseado em modelos que explora o uso combinado de modelos funcionais e de arquitetura durante a concepção

de sistemas embarcados críticos, incluindo os CPS (CORREA et al., 2010). O processo de desenvolvimento sugerido incentiva a exploração do recurso de simulação, oferecido pelas ferramentas modelagem funcional, e a submissão do modelo de arquitetura do sistema a diferentes tipos de análises e verificações, objetivando a geração de um modelo de arquitetura do sistema completo, válido, seguro e confiável.

Como pode ser observado na Figura 3, o processo de desenvolvimento em questão propõe um conjunto e um encadeamento de atividades a serem realizadas durante a concepção de um sistema embarcado crítico.

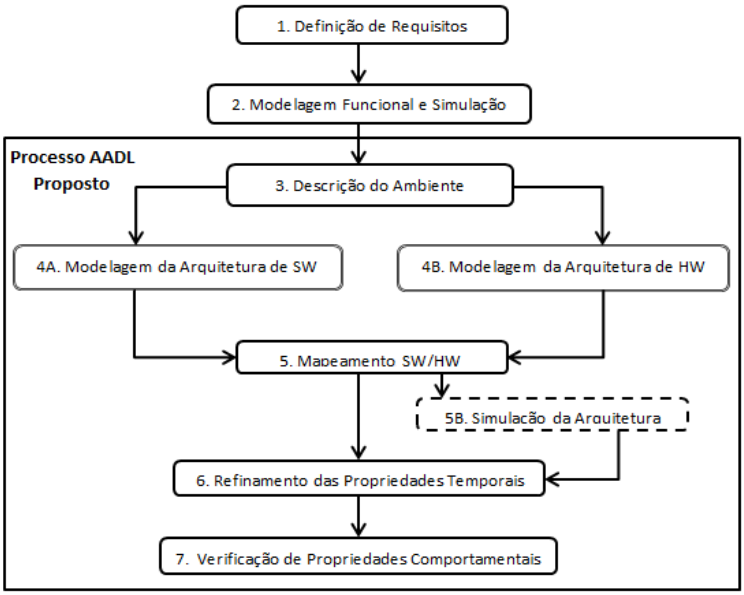

Figura 3 – Encadeamento das Atividades do Processo de Desenvolvimento

Fonte:(CORREA et al., 2010)

De acordo com o fluxo de projeto sugerido pela Figura 3, o processo de desenvolvimento em questão sugere que o primeiro modelo a ser desenvolvido seja o modelo funcional do sistema, o que é feito com o uso de ferramentas de modelagem e simulação, como por exemplo, o Simulink ou o SCADE. Na sequência, é criado uma espécie de diagrama de contexto do sistema, representando a interação do sistema com dispositivos externos, tais como, sensores e atuadores, usando para

isso uma linguagem de descrição de arquitetura, como por exemplo, a AADL. Ainda utilizando uma linguagem de descrição de arquitetura, posteriormente, deve ser realizada a modelagem da arquitetura de software, atividade esta, responsável pelo mapeamento das funcionalidades do sistema (provenientes do Simulink ou do SCADE) para os componentes de software do modelo de arquitetura do sistema. No modelo de arquitetura de software do sistema é possível especificar o comportamento esperado dos componentes do modelo, como *threads* e *devices*, e submeter o modelo resultante a verificação formal de propriedades. Dentre os pontos passíveis de verificação, destacam-se as propriedades comportamentais das threads, tais como vivacidade, ausência de bloqueio e justiça. Posteriormente, ou em paralelo ao projeto da arquitetura de software, deve ser feita a modelagem da arquitetura de hardware, momento em que é definida a plataforma de execução do sistema. Dando continuidade ao processo de desenvolvimento, é feita a integração entre os componentes da arquitetura de software e os componentes da arquitetura de hardware (o chamado *deployment* do software). A partir desse mapeamento é possível submeter o modelo à verificação de propriedades temporais, utilizando para isso, informações como periodicidades, *deadlines* e tempos de execução inseridas no modelo no modelo de arquitetura do sistema.

Entretanto, um dos pontos em aberto neste processo de desenvolvimento é justamente a definição da maneira como se dá a transformação do modelo funcional para um modelo de arquitetura. Neste cenário, a AST aparece como um mecanismo para auxiliar a criação de um modelo de arquitetura de software preliminar a partir de informações estruturais, operacionais (modos de operação) e comportamentais extraídas de um modelo funcional.

1.3 JUSTIFICATIVA

Diferentes processos de desenvolvimento para CPS sugerem o desenvolvimento de um modelo funcional nas fases iniciais do processo de desenvolvimento (veja (DELANGE et al., 2010), (CORREA et al., 2010), (JENSEN; CHANG; LEE, 2011)). Ou seja, o desenvolvimento de um CPS partindo da modelagem funcional combinada com à simulação, utilizando ferramentas de modelagem como o Simulink ou Scade, é considerada interessante, uma vez que esta abordagem de desenvolvimento contribui com a compreensão das funcionalidades do sistema e com o encadeamento das mesmas, isto em uma fase inicial do processo

de desenvolvimento.

Considerando a abordagem de desenvolvimento descrita acima, o mecanismo proposto neste documento oferece suporte para à execução da transição entre a modelagem funcional e a modelagem arquitetural durante o processo de desenvolvimento de CPSs, a qual é considerada por muitos profissionais uma transição de projeto bastante crítica. O mecanismo proposto facilita o estabelecimento de uma ligação entre o modelo funcional (técnicas de controle) e a arquitetura computacional do sistema, buscando garantir desta forma a consistência entre os modelo funcional e de arquitetura.

As abordagens de integração de modelos funcional e de arquitetura que apresentam resultados interessantes concentram-se na utilização de em linguagens de modelagem de alto nível para a modelagem dos requisitos funcionais em nível de aplicação (Simulink ou Scade) e numa linguagem em nível de arquitetura (AADL) para modelar os aspectos não funcionais do sistema. As limitações destas abordagens motivaram e justificam o desenvolvimento do mecanismo de transformação de modelos proposto nesta tese de doutorado. As limitações destas abordagens são descritas em mais detalhes no capítulo 3 deste documento.

1.4 OBJETIVOS

O principal objetivo desta pesquisa de doutorado é contribuir com o processo de desenvolvimento de sistemas embarcados, incluindo CPSs, propondo um mecanismo que ofereça suporte à execução da transformação de um modelo funcional em um modelo de arquitetura preliminar. Uma vez delineado o objetivo geral desta pesquisa, os objetivos específicos são listados abaixo:

1. Sugerir como relacionar os elementos do modelo funcional com os componentes de software do modelo de arquitetura do sistema;

2. Extrair aspectos estruturais (possíveis modos de operação e o comportamento dos componentes) de um modelo funcional para gerar um modelo de arquitetura preliminar do sistema;

3. Diminuir a ocorrência de erros de integração e de inconsistência entre os modelos que orientam o processo de transformação de modelos proposto;

1.5 METODOLOGIA

Este trabalho de pesquisa foi desenvolvido em 4 etapas principais. Na primeira etapa foi feita uma fundamentação teórica, nela buscou-se compreender o funcionamento da técnica de transformação de modelos proposta pela MDE, compreender o princípio de funcionamento e identificar características comuns aos modelos gerados por diferentes ferramentas de modelagem funcional e simulação de algoritmos de controle, e também compreender o princípio de funcionamento e identificar características comuns aos modelos de arquitetura de sistemas, especificados utilizando diferentes linguagens de descrição de arquitetura.

Na segunda etapa foi feito um levantamento do estado da arte, o qual buscou identificar e compreender as abordagens voltadas especificamente à transformação ou integração de modelos funcionais Simulink e modelos de arquitetura em AADL durante a concepção de sistemas embarcados. Uma vez que a ferramenta Simulink e a linguagem AADL foi a combinação de ferramenta de modelagem funcional e de linguagem de descrição de arquitetura definida para gerar, respectivamente, o modelo fonte e o modelo alvo do processo de transformação de modelos proposto por este trabalho de pesquisa.

Na terceira etapa foi feita a estruturação propriamente dita do mecanismo proposto neste trabalho de pesquisa, o qual visa auxiliar a execução da transição de projeto entre a modelagem funcional e a modelagem arquitetural durante o processo de desenvolvimento de CPSs. Nesta etapa também foi implementado o protótipo de uma ferramenta que automatiza o processo de transformação de modelos proposto.

Na quarta etapa foi feita a avaliação do mecanismo proposto. A referida avaliação foi baseada no desenvolvimento de alguns estudos de caso. Dentre os quais, dois deles são apresentados com mais detalhes no decorrer deste documento. Um dos estudos de caso apresentados neste documento utilizou como entrada o modelo funcional Simulink de um sistema embarcado que realiza manobras de estacionamento de forma autônoma (SIAMES). Neste caso, foi possível avaliar a execução das regras de transformação responsáveis pelos mapeamentos estrutural, dos modos de operação e comportamental propostas pela AST. A fim de verificar se o mecanismo proposto possuía capacidade de operar modelos mais complexos, foi utilizado como estudo de caso o modelo funcional Simulink de um Veículo Aéreo Não Tripulado (VANT). Os modelos funcionais Simulink do SIAMES e do VANT foram dispo-

nibilizados pelo grupo de pesquisa em Desenvolvimento de Sistemas Embarcados Críticos do Departamento de Automação e Sistemas da Universidade Federal de Santa Catarina.

1.6 ORGANIZAÇÃO DO DOCUMENTO

Este documento de tese está organizado da seguinte maneira:

Capítulo 2: apresenta uma visão geral dos conceitos básicos usados durante o desenvolvimento desta teste. Ele introduz noções da Engenharia Dirigida por Modelos (MDE), Desenvolvimento Dirigido por Modelos (MDD), Arquitetura Dirigida por Modelos (MDA), além dos conceitos de modelo, metamodelo, metamodelagem, e transformação de modelos. Além disso, ele também apresenta alguns ambientes de desenvolvimento dirigido por modelos e algumas linguagens de descrição de arquitetura, em especial a Linguagem de Projeto e Análise de Arquitetura (AADL), que podem ser utilizados durante o processo de desenvolvimento de CPS.

Capítulo 3: Apresenta as abordagens existentes voltadas especificamente à transformação ou integração de modelos Simulink e modelos AADL durante a concepção de sistemas embarcados.

Capítulo 4: apresenta o mecanismo proposto para transformar um modelo funcional em um modelo de arquitetura de software durante o processo de desenvolvimento de *Cyber-Physical Systems* (CPS). Tal solução é chamada de Transformação Assistida de Modelos (AST).

Capítulo 5: apresenta a aplicação da AST na transformação de modelos funcionais Simulink em modelos de arquitetura AADL.

Capítulo 6: apresenta o desenvolvimento de um estudo de caso que foi desenvolvido com o intuito de avaliar a completude e a utilidade da solução proposta, quando aplicada para obter o SCE de um CPS.

Capítulo 7: apresenta as considerações finais, os resultados alcançados e algumas sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma visão geral dos conceitos básicos da Engenharia Dirigida por Modelos (MDE), do Desenvolvimento Dirigido por Modelos (MDD) e da Arquitetura Dirigida por Modelos (MDA). Apresenta também os conceitos de modelo, metamodelo, metamodelagem, e transformação de modelos, todos conceitos usados ao longo deste trabalho de pesquisa. Além disso, são apresentadas algumas das ferramentas de modelagem funcional e linguagens de descrição de arquitetura mais comumente utilizadas durante o processo de desenvolvimento de CPSs.

2.1 ENGENHARIA DIRIGIDA DE MODELOS

A Engenharia Dirigida por Modelos se concentra em criar e explorar modelos ao invés de código fonte durante o processo de desenvolvimento de um sistema. Seu objetivo é aumentar a produtividade, permitindo o desenvolvimento de sistemas complexos por meio de modelos definidos com conceitos, que são menos ligados à tecnologia de implementação e mais próximos do domínio do problema. Isso faz com que os modelos sejam mais fáceis de especificar, entender e manter, facilitando a compreensão de problemas complexos e suas possíveis soluções através de abstrações (SELIC, 2003).

De acordo com (SCHMIDT, 2006) a MDE combina: (i) linguagens de domínio específico (DSLs), que é a utilização de uma linguagem adequada ao domínio de aplicação; e (ii) motores de transformação, que é a aplicação de sucessivas transformações a partir de modelos iniciais. Estes motores de transformação podem gerar diferentes tipos de artefatos que implementam a aplicação desejada. Existem dois tipos de transformações, à saber: (i) transformação de modelo para modelo (M2M); e (ii) transformação de modelo para código (M2T). Transformações M2M permitem a replicação de elementos em diferentes pontos de determinados modelos, assim como a replicação de elementos dentro de um mesmo modelo. Transformações M2T geram um código fonte a partir de um modelo de especificação.

Processos de desenvolvimento dirigidos por modelos (MDD) são propostos para aplicar de forma estruturada os princípios da MDE, cuja principal meta é a utilização de modelo(s) como o principal artefato a ser manipulado durante a concepção de um sistema. A OMG

(*Object Management Group*) definiu um conjunto de padronizações que possibilita a definição de abordagens de desenvolvimento, orientado por modelos independentes de plataforma. Este conjunto de padronizações é conhecido como Arquitetura Dirigida por Modelos (MDA - *Model Driven Architecture*), e tem como foco principal a separação entre a especificação das funcionalidades e a especificação da implementação em uma plataforma específica (OMG, 2003). Abordagens baseadas em MDA permitem o desenvolvimento de software através da modelagem e aplicação de mapeamentos automáticos entre os modelos e suas respectivas implementações. O objetivo da OMG com a MDA é permitir que os desenvolvedores se concentrem em determinar os requisitos do sistema, não se preocupando com a plataforma em particular onde serão implementados. A MDA adota a UML como linguagem de modelagem padrão. Independentemente do nível de abstração, a MDA classifica as linguagens de modelagem como: (i) linguagens independentes de plataforma, que são orientadas à especificação do sistema; e (ii) linguagens dependentes de plataforma, que são orientadas à implementação do sistema. Ela também aplica o conceito de mapeamento, por meio de mecanismos automáticos ou semi-automáticos para transformar Modelos Independentes de Plataforma (PIM) em Modelos Dependentes de Plataforma (PSM). As linguagens utilizadas para expressar os modelos PIM e PSM são definidas por meio de metamodelos, que são modelos capazes de representar tanto a sintaxe abstrata quanto a concreta, quanto a semântica operacional de cada linguagem de modelagem.

O Eclipse Modeling Framework (EMF) também é considerado uma abordagem MDE bastante significativa. Ele fornece um framework de modelagem e geração de código para aplicações Eclipse baseado em modelos de dados estruturados. Ele é um framework para descrever modelos de classe e geração de código Java, suporta a criação, modificação, armazenamento e carregamento de instâncias desse modelo. Além disso, fornece geradores para apoiar a edição de modelos EMF. O EMF unifica três importantes tecnologias: Java, XML e UML. Independentemente de qual é usada para definir o modelo, um modelo EMF pode ser considerado como a representação comum que incorpora as outras. O núcleo do framework EMF possui um metamodelo para descrever os modelos, ou seja, para definir a estrutura dos modelos que os desenvolvedores usam para manter os dados da aplicação. O Graphical Modeling Framework (GMF) permite que os modelos, ora gerenciados através do EMF, sejam manipulados de forma gráfica (ELIPSE, 2012).

A MDA e o EMF são abordagens MDD representativas que possuem como objetivo comum, a meta de produzir um modelo que contém detalhes suficientes para geração automática ou semi-automática de código executável. Essas abordagens baseiam-se em um conjunto de princípios que envolvem os conceitos de modelo, metamodelo, metamodelagem e transformação de modelos. Como alguns destes conceitos podem ter significados diferentes em outros contextos, as subseções seguintes apresentam informações mais detalhadas sobre os mesmos.

2.1.1 Modelos e Metamodelos

O desenvolvimento de software orientado a modelos está centrado na utilização de modelos. Modelos são abstrações do sistema ou do seu ambiente, e permitem aos desenvolvedores ou as partes interessadas abordar de forma eficaz os requisitos do sistema. Além disso, os modelos são mais baratos para construir do que um sistema real. Por exemplo, engenheiros civis criam modelos estruturais estáticos e dinâmicos de pontes para verificar a segurança estrutural, uma vez que a modelagem é certamente mais barata e mais eficaz do que a construção de pontes reais, para ver em que situações elas vão entrar em colapso (CZARNECKI; HELSEN, 2006). Para (CETINKAYA; VERBRAECK, 2011), os modelos representam diferentes visões do sistema

A MDE busca estimular o uso de linguagens de modelagem como linguagens de programação, e não apenas como linguagens de projeto para gerar documentação. Com isso, os modelos deixam de ser apenas artefatos de documentação para serem os principais artefatos do processo de desenvolvimento de software. Para isso, os modelos necessitam de uma maior precisão para serem parte direta do processo de produção de software. No escopo da MDE, os modelos são os artefatos de entrada para a construção de um sistema através da execução de transformações (MAIA, 2006).

De acordo com (RUSCIO, 2007), na MDE os modelos precisam ser entendidos e manipulados automaticamente. Neste cenário, a metamodelagem desempenha um papel fundamental, uma vez que ela se apresenta como uma técnica para definir a sintaxe abstrata de modelos e as inter-relações entre os elementos do modelo. A metamodelagem pode ser vista como a construção de um conjunto de "conceitos" (coisas, termos, etc) dentro de um determinado domínio. Um modelo é uma abstração dos fenômenos do mundo real, e um metamodelo é mais uma abstração, que realça as propriedades do próprio modelo. Um modelo

está em conformidade com seu metamodelo, assim como um programa está em conformidade com a gramática da linguagem de programação em que ele está escrito (BÉZIVIN, 2005).

Para (MELLOR et al., 2004), um metamodelo normalmente é representado como um diagrama de classes UML, ele define a estrutura, a sintaxe e as restrições para uma família de modelos, isto é, grupos de modelos que compartilham uma sintaxe e uma semântica comuns. Enfim, um metamodelo é uma definição precisa das construções e regras necessárias para criar modelos semânticos, e sua estrutura é uma consequência das relações entre seus elementos.

A OMG apresentou uma arquitetura de metamodelagem em quatro níveis, como ilustrado na figura 4. Na parte inferior, o nível M0 representa o sistema real. Um modelo que representa este sistema fica no nível M1. Este modelo está em conformidade com o seu metamodelo definido no nível M2, e o metamodelo em si, está em conformidade com o metametamodelo do nível M3. O metametamodelo representado no nível M3 está em conformidade consigo mesmo. Esta arquitetura de metamodelagem também é comum a outros espaços tecnológicos, por exemplo, a organização das linguagens de programação. As relações entre documentos XML e esquemas XSD (XML Schema Definition) também seguem os mesmos princípios descritos na Figura 4.

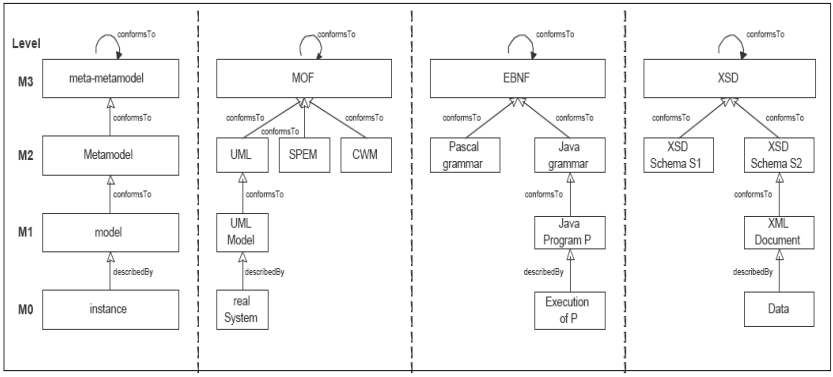

Figura 4 – Os quatro níveis da arquitetura de metamodelagem
Fonte:(RUSCIO, 2007)

2.1.2 Linguagens de Metamodelagem

A metamodelagem é amplamente aceita como uma forma de descrever modelos no contexto da MDE. Um metamodelo é a definição de uma linguagem de modelagem, que por sua vez, também deve ser definido com uma linguagem de modelagem (SPRINKLE et al., 2010).

Uma linguagem de modelagem consiste de uma sintaxe abstrata, uma sintaxe concreta e de uma semântica concreta. A sintaxe abstrata descreve o vocabulário de conceitos fornecidos pela linguagem de modelagem, e como eles podem ser conectados para criar os modelos. Trata-se dos conceitos, relações e regras de boa formação. As regras de boa formação definem como os conceitos podem ser legalmente combinados. Por outro lado, a sintaxe concreta apresenta um modelo na forma de um diagrama (sintaxe visual), ou na forma textual estrutural (sintaxe textual). Já a semântica concreta de uma linguagem de modelagem explica o que a sintaxe abstrata realmente significa (HAREL; RUMPE, 2004).

A metamodelagem é o processo de especificação completo e preciso de uma linguagem de modelagem de domínio específico, que por sua vez, pode ser utilizada para definir os modelos de tal domínio (ACHILLEOS; GEORGALAS; YANG, 2007). Uma linguagem de domínio específico (DSL) é uma linguagem de modelagem dedicada a um domínio específico. As DSLs estão mais próximas do domínio do problema e conceitos do que as linguagens de modelagem de propósito geral, como a UML.

Um metamodelo descreve a sintaxe abstrata de uma linguagem de modelagem, ele é definido por meio de uma linguagem de metamodelagem. Uma linguagem de metamodelagem é uma linguagem usada para descrever novas linguagens de modelagem. O metamodelo de uma linguagem de metamodelagem é chamado de metametamodelo. Para (CETINKAYA; VERBRAECK, 2011), modelo, metamodelo, e metametamodelo formam um padrão de modelagem de três níveis, semelhantes ao apresentado na primeira coluna da figura 4.

Existem várias linguagens de metamodelagem disponíveis atualmente, porém as mais comumente utilizadas são *Facility Meta Object* (MOF), Eclipse ECORE. Sendo que:

- **MOF**: é uma especificação da OMG que define uma linguagem abstrata para a descrição de modelos, mais especificamente, uma linguagem usada para descrever metamodelos. MOF permite especificar formalmente uma linguagem de modelagem. Um

exemplo bastante conhecido da descrição de modelos utilizando MOF é o metamodelo da Linguagem de Modelagem Unificada (UML). Um padrão de apoio do MOF é o XMI (*XML Metadata Interchange*), que define um formato para troca de dados baseado em XML para os modelos nos níveis M3, M2, ou M1 da arquitetura de metamodelagem em quatro níveis, representados gráficamente na figura 4 (OMG, 2006)

- **ECORE**: é a linguagem de metamodelagem do EMF. De acordo com (BUDINSKY, 2004), no EMF um modelo de núcleo corresponde a um metamodelo. O metamodelo para os modelos de núcleo é chamado Ecore. Isto é, o Ecore define a estrutura do modelo de núcleo, que por sua vez define a estrutura dos modelos usados para manter os dados da aplicação. Assim, Ecore é o metametamodelo para os modelos de núcleo (metamodelos). Se for considerada novamente a figura da arquitetura de metamodelagem em quatro níveis representada da figura 4, Ecore está no nível M3, e os modelos EMF no nível M2. A abordagem de modelagem da linguagem ECORE é semelhante à da linguagem MOF. Na verdade, EMF suporta MOF *Essential* (EMOF), que é parte da especificação OMG MOF 2.0.

2.1.3 Transformação de Modelos

A atividade de transformação de modelos ocupa um papel importante em abordagens MDE, além disso, ela está essencialmente enraizada na engenharia de software (CZARNECKI; HELSEN, 2006). Transformações de modelos são usadas para criar novos modelos com base em informações existentes em todo o processo de desenvolvimento (SENDALL; KOZACZYNSKI, 2003). Em vez de criar modelos a partir do zero durante as diferentes fases do ciclo de vida e atividades de desenvolvimento, o processo de transformação de modelos permite a reutilização de informações que já tenham sido modeladas uma vez.

De acordo com (MENS; GORP, 2006), a transformação de modelos é o processo de conversão de um modelo fonte para um modelo alvo, de acordo com um conjunto de regras de transformação. Uma regra de transformação é constituída por duas partes: um lado esquerdo que acessa o modelo fonte (LHS), e um lado direito que expande o modelo alvo (RHS). Sendo que o modelo fonte precisa estar em conformidade com o metamodelo fonte e o modelo alvo precisa estar em conformidade com o metamodelo alvo. Durante o processo de

transformação de modelos, o modelo fonte permanece inalterado. As regras de transformação podem ser especificadas com uma linguagem de transformação de modelos.

Segundo (KLEPPE; WARMER; BAST, 2003), o processo de transformação de modelos é a geração automática de um modelo alvo a partir de um modelo fonte, de acordo com uma definição de transformação. Uma definição de transformação é um conjunto de regras de transformação que, juntas, descrevem como um modelo na linguagem fonte pode ser transformado em um modelo na linguagem alvo. Uma regra de transformação é uma descrição de como uma ou mais construções na linguagem fonte podem ser transformadas em uma ou mais construções na linguagem alvo. Para a OMG, a transformação de modelos é um processo que converte o modelo de um sistema em outro modelo do mesmo sistema (OMG, 2003). Para (BAUDRY et al., 2006), a transformação de modelos é utilizada para relacionar linguagens.

Em uma transformação de modelos simples, existe um modelo de entrada e um modelo de saída, e ambos devem estar em conformidade com seus respectivos metamodelos. As regras de transformação devem ser definidas respeitando os metamodelos, e executadas por um motor de transformação. Uma transformação de modelos também pode ter vários modelos fonte e alvo, sendo que os metamodelos fonte e alvo podem ser o mesmo em algumas situações. Entretanto, é importante ressaltar que se for necessário executar transformações com múltiplos modelos fonte e vários modelos alvo, não é possível haver mais do que duas linguagens diferentes envolvidas (MENS; GORP, 2006) e (CZARNECKI; HELSEN, 2006).

A Figura 5 apresenta a visão geral dos principais conceitos envolvidos em um processo de transformação de modelos, bem como os relacionamentos entre eles.

De acordo com o processo de transformação de modelos apresentado na figura 5, um modelo fonte **Ma**, que deve estar em conformidade com seu metamodelo **MMa**, é transformado no modelo alvo **Mb**, que deve estar em conformidade com o metamodelo **MMb**. A transformação é definida pelo modelo de transformação (conjunto de regras de tranformação) **Mt**, que deve estar de acordo o metamodelo de transformação **MMt**. O metamodelo de transformação **MMt**, o metamodelo fonte **MMa**, e o metamodelo alvo **MMb** devem, obrigatoriamente, estar em conformidade com metametamodelo **MMM**.

A seguir são apresentados alguns conceitos relacionados ao domínio de transformação de modelos, importantes para o entendimento do mecanismo proposto neste documento tese.

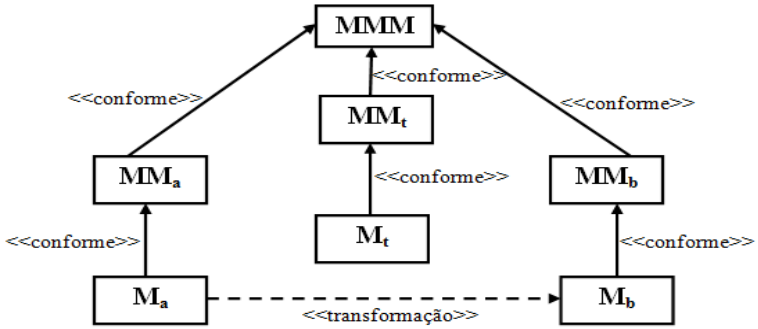

Figura 5 – Visão Geral de um Processo de Transformação de Modelos
Fonte:(LINA; NANTES, 2006)

- **Regras de Transformação**. As regras de transformação baseiam-se essencialmente em mapeamentos. Mapeamentos especificam as correspondências ou inter-relações entre os elementos de dois metamodelos diferentes. O processo de identificação e caracterização das inter-relações entre os elementos de dois metamodelos é chamado de *esquema de correspondência* (RAHM; BERNSTEIN, 2001).

  De acordo com (LOPES et al., 2006), os mapeamentos ligam elementos do modelo fonte a elementos do modelo alvo. A multiplicidade ou cardinalidade destas ligações pode ser do tipo: *um para um*, *um para muitos*, ou *muitos para um*. A Figura 6 exibe uma representação gráfica destes três tipos de multiplicidade de um mapeamento.

  Os mapeamentos são caracterizados por elementos do modelo alvo que representam a mesma estrutura e possuem a mesma semântica de elementos do modelo fonte. Como pode ser observado na Figura 6, o mapeamento do tipo *um para um* **(1:1)** representa que um elemento do modelo alvo possui a mesma semântica de um elemento do modelo fonte. O mapeamento do tipo *muitos para um* **(n:1)** significa que um elemento do modelo alvo possui a mesma semântica de um conjunto de elementos do modelo fonte. O mapeamento do tipo *um para muitos* **(1:n)** representa que um conjunto de elementos do modelo alvo possui a mesma semântica de um elemento do modelo fonte.

  Outra classificação interessante para mapeamentos é a utilizada

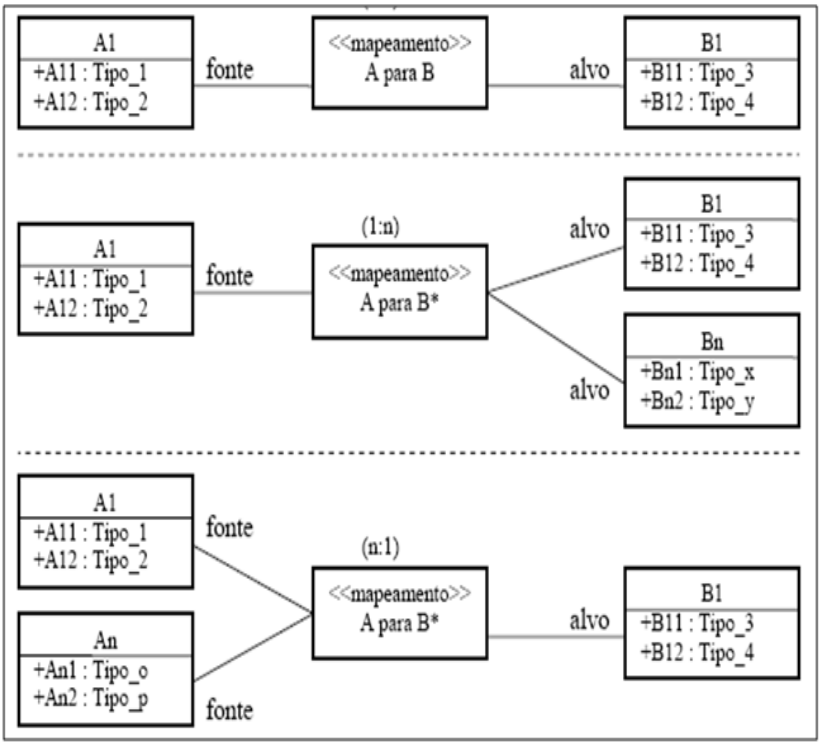

Figura 6 – Categorias de mapeamento (a)um-para-um, (b)um-para-muitos e (c) muitos-para-um
Fonte: (LOPES et al., 2006)

pelo *framework* MDA da OMG, o qual define três categorias básicas de mapeamentos (OMG, 2003):

1) Mapeamento de **TIPOS**: esta categoria inclui mapeamentos definidos de acordo com os tipos dos elementos dos modelos. Por exemplo, em uma transformação entre um modelo de classe UML e um modelo relacional, um possível mapeamento poderia ser definido entre os elementos do tipo classe e do tipo tabela. Com isso, todas as classes seriam mapeadas como tabelas após a execução da respectiva regra de transformação. O mapeamento de tipos é capaz de expressar apenas regras de transformação que geram tipos de elementos no modelo alvo a partir de tipos de elementos do modelo fonte (por exemplo, transformar classes

em tabelas). Ou seja, neste caso, a transformação é sempre executada de forma determinística e guiada por informações pré estabelecidas.

2) Mapeamento de **INSTÂNCIAS** - nesta categoria, os elementos mapeados possuem marcações que indicam como a transformação será realizada. A diferença para a categoria anterior está no fato de que os elementos, que são instâncias de um mesmo tipo, podem ser mapeados de forma diferente, de acordo com as marcas que eles possuem.

Para realizar o mapeamento de instâncias, a MDA especifica marcas. Marcas representam conceitos do modelo ALVO, que são aplicadas a elementos do modelo fonte para indicar como estes devem ser transformados. Ou seja, as marcas definem quais elementos devem ser criados no modelo alvo a partir de um elemento marcado no modelo fonte (OMG, 2003). O uso das marcas cria um passo adicional no processo de transformação de modelos, ou seja, as marcas devem ser inseridas nos elementos do modelo FONTE antes das regras de transformação serem executadas, criando assim um modelo intermediário. O novo modelo fonte, o qual que possui as marcas, é chamado de *modelo fonte marcado*. O *modelo fonte marcado* deve ser utilizado como entrada no processo de transformação de modelos para gerar o modelo alvo. A figura 7 é uma adaptação da Figura 5, e representa graficamente o papel das marcas em um processo de transformação de modelos.

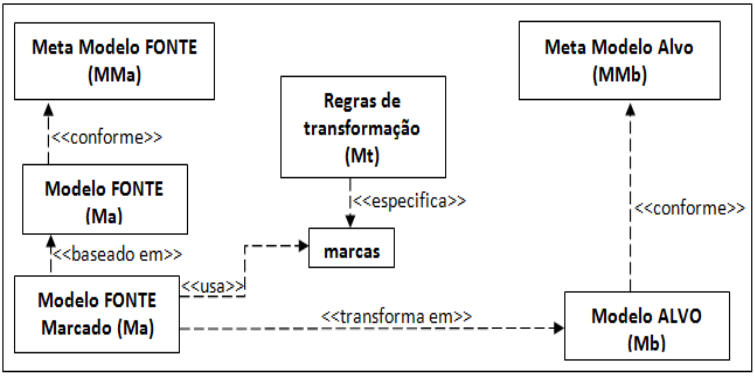

Figura 7 – Mapeamento de Instâncias com marcas

Para serem corretamente utilizadas, as marcas precisam ser estruturadas, restritas e modeladas. Isso porque, determinadas marcas podem ser aplicadas somente a certos elementos do modelo fonte, enquanto outras podem ser mutuamente exclusivas

3) Mapeamento **COMBINADO**: Este tipo de mapeamento combina características dos dois tipos de mapeamento já citados. Assim, um mapeamento de tipos, pode ser, de certa forma, configurado através do uso de marcas. Neste caso, as marcas indicam características presentes no modelo ALVO que não poderiam ser definidas através dos tipos de elementos existentes no modelo FONTE.

- **Tipos de Transformação de Modelos**. De acordo com (MENS; GORP, 2006), as transformações de modelo podem ser classificadas de acordo com as linguagens de modelagem envolvidas e de acordo com o nível de abstração dos modelos.

  Com base na linguagem em que os modelos de origem e de destino de uma transformação são expressos, pode ser feita uma distinção entre transformações endógenas e exógenas. Transformações endógenas são transformações entre modelos expressos na mesma linguagem. Transformações exógenas são transformações entre modelos expressos usando diferentes linguagens. Ou seja, se os metamodelos de origem e de destino são idênticos, a transformação é chamada de endógena, caso contrário, ela é chamada de exógena.

  Uma transformação horizontal é uma transformação onde os modelos de origem e de destino residem no mesmo nível de abstração. Um exemplo típico é a refatoração. Uma transformação vertical é uma transformação em que os modelos de origem e de destino residem em diferentes níveis de abstração. Um exemplo típico é o refinamento, onde uma mesma especificação é gradualmente refinada em uma implementação de pleno direito, por meio de passos de refinamento sucessivos que adicionam detalhes mais concretos.

  A Tabela 1 mostra que as dimensões horizontais *vs* verticais e engógenas *vs* exógenas são verdadeiramente ortogonais, dando um exemplo concreto das combinações possíveis. Como um esclarecimento para o refinamento formal mencionado na tabela1, uma especificação lógica de predicados de primeira ordem ou teoria de conjuntos pode ser gradualmente refinada de tal forma

que o resultado final usa exatamente a mesma linguagem que a especificação original (por exemplo, pela adição de mais axiomas).

	Horizontal	**Vertical**
Endógenas	Refatoração	Refinamento Formal
Exógenas	Migração de Linguagem	Geração de Código

Tabela 1 – Dimensões Ortogonais das Transformações de Modelo
Fonte:(MENS; GORP, 2006)

De acordo com (CZARNECKI; HELSEN, 2006), em alto nível, é possível identificar duas categorias principais de abordagens de transformação de modelo, sendo elas: (i) transformação de modelo para modelo (M2M), e (ii) transformação modelo para texto (M2T). A diferença entre elas é que, enquanto uma transformação M2M cria seu modelo destino como uma instância do metamodelo de destino, uma transformação M2T gera essencialmente *strings*.

Abordagens M2T são úteis para a geração de código executável e artefatos de documentação. Em geral, é possível considerar as transformações M2T como um caso especial de transformações M2M, desde que seja fornecido um metamodelo da linguagem de programação alvo. No entanto, por razões práticas de reutilização de tecnologia de compilação existente e simplicidade, o código é muitas vezes gerado como texto, que é então introduzido e um compilador.

Na categoria M2T, é possível distinguir entre abordagens baseada em *Visitor* (*Visitor-Based Approaches*) e abordagens baseadas em *Templates*. Na categoria M2M é possível distinguir entre as abordagens de manipulação direta, orientadas a estrutura, operacional, baseadas em *template*, relacional, baseadas em transformações gráficas, e híbridas.

- **Critérios de Avaliação para Transformações de Modelos**. De acordo com (CETINKAYA; VERBRAECK, 2011), o objetivo das transformações de modelo é gerar automaticamente diferentes visões de um sistema a partir de um modelo fonte, e suportar a geração de código. Tanto o modelo de destino, quanto as regras de transformação podem ser avaliadas de acordo com alguns princípios básicos derivados da engenharia de software. Sendo assim, alguns critérios de avaliação de transformações de

modelo sugeridos por (CETINKAYA; VERBRAECK, 2011), os quais estendem o trabalho de (MENS; GORP, 2006) são os seguintes:

- **Correção:** A correção de uma transformação de modelos é analisada de duas maneiras: a correção sintática e a semântica (MENS; GORP, 2006). Se o modelo de destino está em conformidade com a especificação do metamodelo alvo, então a transformação do modelo é sintáticamente correta. Se a transformação do modelo preserva o comportamento do modelo de origem, então é semânticamente correta (EHRIG; ERMEL, 2008).
- **Completude:** A fim de preservar e reutilizar as informações no modelo fonte, a transformação precisa ser completa. Se para cada elemento do modelo fonte houver um elemento correspondente no modelo alvo, então a transformação de modelos é completa (MENS; GORP, 2006).
- **Singularidade:** Se a transformação de modelos gera modelos de destino exclusivos para cada modelo fonte, então ela fornece singularidade;
- **Terminação:** Se a transformação de modelos sempre termina e leva a um resultado, consequentemente ela fornece terminação.
- **Legibilidade:** Se as regras de transformação são legíveis e compreensíveis, então a transformação de modelos fornece legibilidade.
- **Eficiência:** As transformações de modelos precisam ser executadas de forma eficiente. A eficiência da transformação pode ser avaliada através da análise de quantos passos de transformação são necessários, e quantas funções são aplicadas durante cada transformação específica.
- **Manutenção:** O grau de esforço despendido para a mudar, amplicar ou reaplicar a transformação de modelos define a manutenção.
- **Escalabilidade:** A capacidade de lidar com grandes modelos sem sacrificar o desempenho define a escalabilidade (MENS; GORP, 2006).
- **Usabilidade:** Uma vez implementada, se a transformação de modelos pode ser facilmente executada, então ela é utilizável.

- **Reutilização:** Reutilização pode ser medida com a possibilidade de adaptação e reutilização de uma transformação de modelos através de vários mecanismos de reutilização, como parametrização ou o uso de templates (MENS; GORP, 2006).
- **Precisão:** Se todos os possíveis erros são tratados, e se a transformação de modelos consegue gerenciar os modelos de origem incompletos, então ela oferece precisão.
- **Robustez:** Se a maioria dos erros inesperados podem ser tratados, e se a transformação de modelo consegue gerenciar os modelos de origem inválidos, então ela fornece robustez. A diferença em relação à Precisão é que um modelo fonte incompleto pode ser visto pela ferramenta de modelagem associada, ao passo que um modelo fonte inválido não pode ser visualizado pela ferramenta de modelagem associada.
- **Modularidade:** A modularidade pode ser definida como a capacidade de combinar transformações existentes em novos compostos. A decomposição de uma transformação complexa em pequenos passos pode exigir um mecanismo para especificar como essas transformações menores devem ser combinadas (SENDALL; KOZACZYNSKI, 2003).
- **Validade:** Uma vez que as transformações podem ser consideradas como um tipo especial de programa de software, as técnicas de teste e de validação podem ser aplicadas nas transformações para assegurar que elas têm o comportamento desejado (MENS; GORP, 2006).
- **Consistência:** Transformações de modelos precisam ser consistentes e inequívocas. Se a transformação de modelos detecta e possivelmente resolve as contradições internas e inconsistências, então ela é consistente (MENS; GORP, 2006).
- **Parcimônia:** Um modelo de destino deve ser o mais parcimonioso e tão leve quanto possível. Simplicidade e falta de redundância podem definir o nível de parcimônia.
- **Rastreabilidade:** A rastreabilidade é a propriedade de ter um registro das ligações entre os elementos de origem e de destino, bem como as várias fases do processo de transformação. Ligações de rastreabilidade podem ser armazenadas tanto no modelo de destino quanto separadamente (DEHAYNI et al., 2009).

  - **Reversibilidade:** Um modelo de transformação a partir de S para T é reversível se houver um modelo de transformação de T para S. Esta propriedade pode ser útil para cancelar os efeitos de uma transformação (DEHAYNI et al., 2009).
  - **Nível de Automação:** Pode e deve ser feita uma distinção entre as transformações de modelo que podem ser automatizadas e as transformações que precisam ser executadas manualmente, ou que pelo menos precisam de uma certa quantidade de intervenção manual.

  É fato que existem muitas maneiras de especificar um mecanismo de transformação de modelos. Porém não existe uma regra ou orientação que pode ser fornecida para definir uma boa solução geral. No entanto, com base em critérios bem definidos é possível analisar os mecanismos de transformação de modelos disponíveis, e os resultados podem ser utilizados para definir o melhor mecanismo de transformação de modelos para o problema em questão.

- **Linguagens de Transformação de Modelos**. O uso de linguagens de programação de propósito geral, como Java, para especificar diretamente transformações de modelos é possível, porém, normalmente é mais oneroso do que o uso de linguagens dedicadas à transformação de modelos (MENS; GORP, 2006). Desta forma, muitas linguagens e ferramentas têm sido propostas para especificar e executar programas de transformação. Mesmo a OMG propondo a linguagem de transformação de modelos QVT (Query/View/Transformation) como padrão, a área de transformação de modelos continua a ser um assunto de intensa pesquisa. Nos últimos anos, uma série de abordagens de transformação de modelos têm sido propostas, tanto no meio acadêmico quanto na indústria. Os paradigmas, as construções, as abordagens de modelagem e o suporte de ferramentas, diferenciam-se com uma certa adequação para um determinado conjunto de problemas. (RUSCIO, 2007).

  Existem várias linguagens classificadas como linguagens de transformação de modelos, algumas próprias para transformações do tipo M2M, e outras para transformações do tipo M2T. Considerando que o contexto desta pesquisa é a transformação M2M, são descritas a seguir duas linguagens que têm sido bastante utilizadas neste domínio, a QVT (*Query/View/Transforamtion*) da OMG e a ATL (*Atlas Transformation Language*).

A linguagem **MOF QVT** ou simplesmente **QVT** é uma especificação da OMG dentro do contexto MDA, que tem como objetivo permitir a definição de regras sobre a verificação e transformação entre modelos. Ela permite definir mapeamentos entre modelos de diferentes metamodelos, usando construções MOF. Enfim, através de um programa QVT é possível definir os modelos envolvidos na transformação, o tipo de transformação que será feita, a direção desta transformação (modelo fonte para modelo alvo), os domínios de cada transformação, cláusulas *when/where*, relações entre os modelos de cada domínio, relações de níveis mais altos ou mais baixos, tipos de transformação *check e enforce* onde se pode somente verificar modelos (OMG, 2008).

A linguagem **ATL** (*Atlas Transformation Language*) é uma linguagem de transformação de modelos desenvolvida pelo grupo ATLAS (INRIA e LINA). No domínio da MDE, a ATL possui recursos para produzir e definir modelos de destino a partir de modelos de origem. Ela foi desenvolvida sobre plataforma Eclipse, e seu entorno integrado possui ferramentas que permitem um fácil desenvolvimento de integração relativo a técnicas de transformação entre modelos. Uma vez que a ATL está definida como um padrão Eclipse, se torna mais fácil desenvolver e integrar modelos e metamodelos ECORE à linguagem de transformação ATL, permitindo um trabalho conjunto entre eles. ATL foi criada tendo como conceitos de inspiração a linguagem QVT da OMG, e atualmente fornece suporte às linguagens de metamodelagem MOF e Ecore. Isto significa que a ATL é capaz de manipular metamodelos que foram especificados de acordo com padrão MOF ou com a semântica Ecore (FOUNDATION,).

A Figura 8 é uma adaptação da figura 5, e demonstra o papel das linguagens de metamodelagem e de transformação de modelos durante o processo de transformação de modelos.

2.2 FERRAMENTAS DE MODELAGEM FUNCIONAL

Muitas pesquisas têm sido realizadas no sentido de criar ferramentas e ambientes de desenvolvimento baseados em MDE para serem utilizados no desenvolvimento de CPS. Nestes ambientes, os modelos representam a especificação do sistema, seu comportamento, suas condições de operação, e também a forma como o sistema deve responder ao ambiente em que está inserido. Algumas das ferramentas

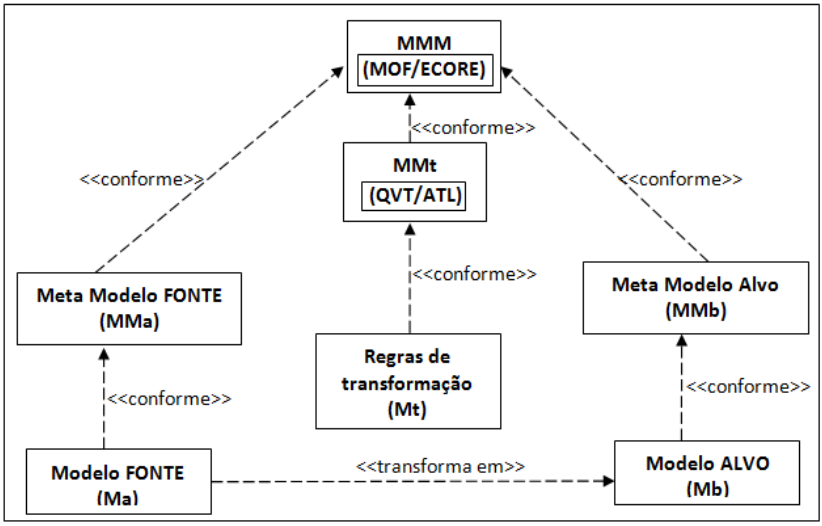

Figura 8 – Papel das Linguagens em um Processo de Transformação de Modelos

que aplicam princípios da MDE, e que são comumente utilizadas pela indústria e também exploradas pela comunidade acadêmica para o desenvolvimento de CPS, são apresentadas brevemente a seguir.

- **MATLAB/Simulink/Stateflow**. MATLAB é uma linguagem de alto desempenho para cálculos numéricos desenvolvida e distribuída pela empresa *The MathWorks*. O nome é derivado de *Matlab MATrix Laboratory*. Ela integra computação, visualização e programação em um ambiente de desenvolvimento onde problemas e soluções são expressos em notação matemática.

  O Simulink é uma ferramenta de modelagem gráfica complementar para o MATLAB, que proporciona à linguagem MATLAB a capacidade de projetar e simular sistemas dinâmicos e lineares em um ambiente gráfico. Ele pode trabalhar com sistemas lineares, não-lineares, *continuous-time*, tempo discreto, multivariável e *multirate*.

  A maioria dos CPS encontrados na vida real são sistemas híbridos, ou seja, apresentam comportamento contínuo e discreto. Sistemas híbridos podem ser modelados, simulados e analisados utilizando Simulink. Um modelo Simulink é especificado textualmente por

meio de uma linguagem de marcação própria da *MathWorks*, e armazenado em um arquivo de texto ASCII em um formato específico. Os arquivos de modelos Simulink possuem a extensão ".mdl".

Os modelos Simulink de sistemas dinâmicos são representados graficamente como diagramas de blocos, compostos por blocos que representam um subsistema primário dinâmico, e por linhas que representam conexões entre as entradas e saídas dos blocos. Cada bloco é uma caixa preta no sentido clássico, com um conjunto de entradas e saídas, e um conjunto de estados que são internos ao bloco. Intrínseco ao bloco, está um conjunto de funções matemáticas que especifica as relações tempo-dependente entre suas entradas, estados internos e saídas. A natureza exata dessa relação matemática não está disponível em um arquivo mdl, no entanto, um tipo de bloco exclusivo no arquivo mdl serve como referência. Propriedades chave de blocos padrões são parametrizadas. Os valores dos parâmetros são constantes ou expressões MATLAB. Os parâmetros para todos os blocos estão disponíveis como pares *nome/valor* no arquivo mdl. A ferramenta Simulink suporta a composição hierárquica de um sistema complexo por meio de subsistemas, cada qual composto por um diagrama de blocos (MATHWORKS, 2011).

O Stateflow é uma ferramenta de design gráfico interativo que trabalha com Simulink para modelar e simular sistemas orientados por eventos, também chamados de sistemas reativos. Ele fornece um editor gráfico, no qual os objetos gráficos do Stateflow contidos na paleta de projeto podem ser utilizados para criar máquinas de estado finito. Um bloco Stateflow é tratado como um bloco padrão do Simulink e, portanto, pode ser inserido normalmente em modelos Simulink. Ou seja, dados podem ser transmitidos de blocos Simulink para gráficos Stateflow, e vice-versa.

- **SCADE Suite**. É um ambiente de desenvolvimento de aplicações críticas, ele fornece uma entrada gráfica e um ambiente de simulação que permitem a representação visual e intuitiva de sistemas de controle. Os requisitos do sistema também são claramente representados no SCADE. Testes funcionais podem ser realizados através de simulações, e os recursos de verificação formal disponíveis no SCADE Suite permitem a validação do comportamento do projeto e a detecção precoce de erros. SCADE

também gera automaticamente código C e ADA legível, rastreável e confiável. O processo de geração de todo o código é compatível com o padrão para aviação civil DO-178B. O KCG, que é o gerador de código do SCADE Suite, é classificado de acordo com o padrão DO-178B, como sendo de Nível A, eliminando assim a necessidade de verificação do código gerado. Ao automatizar a validação e a geração de código embarcado o SCADE permite a detecção precoce de erros de projeto e reduz o tempo e o custo de desenvolvimento.

O SCADE Suite fornece suporte à integração de modelos Simulink com modelos criados no SCADE, e tal integração é suportada pelo *SCADE Simulink Gateway*. Ambos, Simulink e SCADE permitem a modelagem e simulação de sistemas de controle, e, também, permitem uma representação de fluxo de dados. Eles possuem mecanismos semelhantes que facilitam a captação e representação gráfica de sistemas de controle, mas há diferenças importantes entre eles, uma vez que, originalmente, o Simulink é uma ferramenta de modelagem matemática, e não uma ferramenta de desenvolvimento de software, e que o SCADE por outro lado, foi concebido como uma ferramenta de especificação e implementação de software. A principal vantagem de integrar modelos Simulink com modelos SCADE é aumentar a produtividade das equipes de desenvolvimento, uma vez que o SCADE oferece recursos que podem tornar os projetos desenvolvidos em Simulink mais seguros, permitindo, por exemplo, a geração de código potencialmente certificado, exato, seguro e que atende aos requisitos do projeto do software (TECHNOLOGIES, 2011).

- **LabVIEW**. O ambiente de programação gráfico LabVIEW (*Laboratory Virtual Instrument Engineering Workbench*), projetado pela empresa *National Instruments*, é utilizado principalmente para desenvolver sistemas de teste, medição e controle. Ele oferece suporte à linguagem de programação gráfica baseada em fluxo de dados LabVIEW. Nesta linguagem, o fluxo de dados determina a execução do sistema (WIKIPEDIA, 2013).

  Utilizando o LabVIEW é possível interagir com o hardware para a medição e controle, analisar dados, compartilhar resultados e distribuir sistemas. O LabVIEW oferece funções para medição e análise, possibilidade de execução em multiplataformas e dispositivos embarcados. Ele oferece suporte à comunicação com diversos hardwares industriais, como GPIB, VXI, PIX, RS-232, RS-485 e

dispositivos DAQ plug-in, e também possui recursos internos para conectar uma aplicação à Internet, utilizando LabVIEW Web Server e aplicativos ActiveX e redes TCP/IP (CORPORATION, 2013).

Programas LabVIEW são chamados de instrumentos virtuais, ou VIs, isso porque sua aparência e operação imitam instrumentos físicos, como osciloscópios e multímetros. Cada VI é composto por três partes principais a saber: o painel frontal ("panel"), o diagrama de blocos ("diagram") e o ligador de ícones. O painel frontal é a interface com o usuário que permite introduzir e/ou fixar valores em um sistema, e depois verificar os seus efeitos e saídas no diagrama. Devido ao fato do painel frontal ser muito semelhante ao painel frontal de um aparelho de medida verdadeiro, as entradas são chamadas de controles, e as saídas são chamadas de indicadores. É possível utilizar uma grande variedade de controles e indicadores, tais como: interruptores, botões, telas gráficas, etc. Tudo isto para projetar um painel frontal com comandos facilmente identificáveis e compreensíveis. Cada painel frontal tem o seu diagrama de blocos correspondente (CINEL, 2004).

O diagrama de blocos é construído usando a linguagem gráfica de programação G, e pode ser entendido como um código-fonte (contém o código gráfico do programa). Os componentes do diagrama de blocos constituem os nós do programa, por exemplo, estruturas como "Loops" e "Cases", ou ainda funções aritméticas. Os componentes ou estruturas são ligados através de traços que podem representar fios condutores ou simplesmente um fluxo de dados. É possível utilizar o conector de ícones para transformar um VI em um componente de outro VI, neste caso um sub-VI, que pode ser considerado como sendo um sub-programa de um diagrama de blocos de outro VI. Um ícone inserido graficamente representa o VI (de escopo inferior) que foi inserido no diagrama de blocos do VI principal ou superior. Os terminais do ícone indicam onde se deve ligar as entradas e saídas do ícone. Os terminais do sub-VI comportam-se como sendo parâmetros de um sub-programa, e correspondem aos controles e indicadores do painel frontal do VI. A versatilidade do Labview reside essencialmente na modularidade dos VIs e nas relações de hierarquia que é possível estabelecer entre eles. Após o usuário ter criado um VI, ele pode invocá-lo e usá-lo como sendo um sub-VI no diagrama de outro VI de escopo superior. Não existe limite para o número

de níveis hierárquicos dos VIs. Entretanto, em projetos de grande dimensão é muito importante planejar a estrutura do VI principal desde o início ((WIKIPEDIA, 2013), (CINEL, 2004)).

- **Modelica**. De acordo com (FRITZSON, 2010) e (ASSOCIATION, 2012), a linguagem Modelica é uma linguagem de modelagem declarativa, orientada a objetos e multi-domínio que permite a especificação de modelos matemáticos de sistemas físicos grandes, complexos e heterogêneos para fins de simulação computacional. Entre os sistemas físicos que podem ser modelados com a linguagem Modelica estão os sistemas mecânicos, elétricos, hidráulicos, térmicos e de controle. Modelica é uma linguagem *free* e de código aberto desenvolvida pela organização sem fins lucrativos *Modelica Association*, a qual também desenvolveu a biblioteca padrão Modelica que contém centenas de modelos de componentes genéricos e funções em vários domínios.

  Do ponto de vista do usuário, os modelos são descritos por esquemas, também chamados de diagramas de blocos. Um esquema consiste de componentes ligados. Um componente tem "conectores" (muitas vezes também chamado de "portas") que descrevem as possibilidades de interação, como por exemplo, um pino elétrico, ou um sinal de entrada. Ao desenhar linhas de conexão entre os conectores um sistema físico ou diagrama de blocos é construído. Internamente um componente é definido por um outro esquema ou no nível "baixo´´, por uma descrição da equação base do modelo na sintaxe Modelica.

  A linguagem Modelica é uma descrição textual utilizada para definir todas as partes de um modelo e estruturar os componentes do modelo em bibliotecas, chamadas de pacotes. Um ambiente de simulação Modelica apropriado é necessário para editar graficamente uma especificação, realizar simulações e outras análises. Informações sobre tais ambientes estão disponíveis no *www.modelica.org/tools.* Basicamente, todos os elementos de linguagem Modelica são mapeados em equações diferenciais, algébricas e discretas.

- **Scilab**. De acordo com (CAMPBELL; CHANCELIER; NIKOUKHAH, 2006) e (MA; XIA; PENG, 2008), o Scilab é um software científico para computação numérica semelhante ao Matlab que fornece um ambiente computacional aberto para aplicações científicas. Em desenvolvimento desde 1990 pelos pesquisadores do INRIA (*Institut National de Recherche en Informatique et en Automatique*) e

do ENPC (*École Nationale des Ponts et Chaussées*), é distribuído gratuitamente e em código aberto através da Internet desde 1994. A partir de 2003, passou a ser de responsabilidade do Consórcio Scilab, e desde julho de 2012 é mantido e desenvolvido pelo Scilab Enterprises.

Scilab é uma linguagem de programação de alto nível, orientada à análise numérica. Possui funções especializadas para computação numérica, organizadas em bibliotecas chamadas de caixas de ferramentas que cobrem áreas como simulação, otimização, sistemas de controle e processamento de sinais. Estas funções reduzem consideravelmente a carga de programação em aplicações científicas.

A linguagem Scilab possui um ambiente de modelagem e simulação de sistemas conhecido como Scicos, ele fornece um editor gráfico de diagrama de blocos para a construção e simulação de sistemas dinâmicos. Scilab/Scicos é a única alternativa de código aberto para ferramentas comerciais de modelagem e simulação de sistemas dinâmicos, como MATLAB/Simulink e MATRIXx/SystemBuild. Scilab/Scicos é particularmente útil em processamento de sinais, sistemas de controle, estudo de filas, física e sistemas biológicos. Ele permite ao usuário modelar e simular a dinâmica de sistemas dinâmicos híbridos através da criação de diagramas de blocos, usando um editor baseado em GUI, e compilar modelos em códigos executáveis. Há um grande número de blocos padrão disponível nas paletas. Também é possível que o usuário programe novos blocos em C, Fortran, ou Linguagem Scilab, e construa uma biblioteca de blocos reutilizáveis que podem ser utilizados em sistemas diferentes. Scilab/Scicos permite a execução de simulações em tempo real e geração de código C a partir do modelo Scicos usando um gerador de código.

2.3 LINGUAGENS DE DESCRIÇÃO DE ARQUITETURA

De acordo com (CHKOURI; BOZGA, 2009), uma linguagem de descrição de arquitetura (ADL) é uma linguagem que oferece recursos para a modelagem da arquitetura conceitual de um sistema de software, e se distingue da implementação do sistema. ADLs fornecem uma sintaxe concreta e uma estrutura conceitual para especificar arquiteturas. Os blocos de construção de uma ADL são componentes, conectores,

e configurações arquitetônicas. Um componente em uma arquitetura é uma unidade de computação ou um repositório de dados. Os conectores são links arquiteturais usados para modelar as interações entre os componentes e as regras que regem essas interações. As configurações arquitetônicas ou topologias, são gráficos de componentes e conectores que descrevem uma estrutura arquitetônica. Esta informação é necessária para determinar se os componentes estão apropriadamente conectados, se suas interfaces (conectores) permitem uma comunicação adequada, e se sua semântica combinada resulta no comportamento desejado para o sistema. Enfim, uma ADL deve modelar de forma explícita os componentes, os conectores e suas configurações, além disso, para ser verdadeiramente usável e útil, ela deve possuir suporte de ferramenta(s).

O termo ADL tem sido utilizado no contexto da concepção de ambas arquiteturas, de software e de hardware. ADLs de software são usadas para representar e analisar arquiteturas de software, elas capturam as especificações de comportamento dos componentes e suas interações compõem a arquitetura do software. No entanto, ADLs de hardware capturaram a estrutura (componentes de hardware e sua conectividade), e o comportamento (conjunto de instruções) de arquiteturas de processadores (MISHRA; DUTT, 2005). Foram propostas várias ADLs para a modelagem de arquiteturas, tanto dentro de um domínio particular, como linguagens de modelagem de arquietura de uso geral. Algumas das linguagens comumente referidas como ADL são *C2* (MEDVIDOVIC et al., 1996), (MEDVIDOVIC; ROSENBLUM; TAYLOR, 1999), *Rapide* (LUCKHAM et al., 1995), *Darwin* (MAGEE; KRAMER, 1996), *UniCon* (SHAW et al., 1995), *SADL* (MORICONI; RIEMENSCHNEIDER, 1997), (CRETTAZ et al., 2001)), AADL (SAE, 2011b).

De acordo com (CHKOURI; BOZGA, 2009), as comparações entre as linguagens apresentadas nas Tabelas 2 e 3 são em relação a componentes, conexões, prioridades entre os componentes, descrição do comportamento e suporte para sistemas embarcados distribuídos.

Todas estas linguagens fazem distinção entre a interface do componente e a instância de um componente que apresenta essa interface, e todas as linguagens fornecem sintaxe e semântica para a especificação da interface do componente. Todas as linguagens exibem uma especificação de interface do componente como sendo a definição de um tipo de componente, podendo haver várias instâncias de componentes que apresentam essa mesma interface. Todas as linguagens também suportam uma composição hierárquica que permite descrever arquiteturas de sistemas de software em diferentes níveis, por meio de um conjunto de

	Componente		
	Interface	**Implementation**	**Propriedades Não-Funcionais**
C2	Portas superiores e inferiores	componente implementation	nenhum
SADL	Portas de entrada e saída(iports e oports	componente implementation	requer modificação de componente
Rapide	ação, serviço	interface, componente implementation	nenhum
Darwin	serviços	componente implementation	nenhum
Unicon	players (atores)	componente implementation	atributos para análise de escalonabilidade
AADL	portas de entrada e saída (event e data); parametros in e out	componente implementation	atributos de restrições de tempo, escalonabilidade, safety

Tabela 2 – Comparação entre ADLs
Fonte: fonte:(CHKOURI; BOZGA, 2009)

subcomponentes e conexões entre esses subcomponentes. Além disso, *C2*, *Darwin*, *SADL* e *UniCon* compartilham muito do seu vocabulário, e se referem a eles simplesmente como componentes; no Rapide eles são interfaces, e em AADL categorias de componente.

AADL é uma ADL que oferece suporte para a especificação de sistemas embarcados, prioridade para a análise de escalonabilidade, comportamento usando máquina de estado e propriedades funcionais e não-funcionais. Além disso, o que é notável sobre a AADL é o seu forte apoio sintático e semântico para arquiteturas que consistem de componentes de um número limitado de categorias funcionais. Junto a isso, ela permite adicionar propriedades não-funcionais a componentes da arquitetura, como o *timing*, o consumo de memória e propriedades de *safety*. Dessa maneira, o modelo de uma arquitetura de sistema permite que ferramentas específicas verifiquem propriedades não funcionais do sistema em fases iniciais do projeto, o que faz com que a notação AADL seja particularmente interessante para o desenvolvimento de software embarcados.

Em comparação a outras linguagens de modelagem, AADL define abstrações de baixo nível, incluindo descrições de hardware. Estas abstrações são mais propensas a ajudar a projetar um modelo detalhado perto do produto final. AADL foi introduzida pela primeira vez para modelar arquiteturas de software e hardware nos domínios aviônico e

	Componente			
	Conectores	**Prioridades**	**Comportamento**	**Distribuído**
C2	Interface com cada componente via uma porta separada.	prioridade alta	Consiste de uma invariante e um um conjunto de operações. A invariante é usada para especificar propriedades que devem ser verdadeiras em todos os estados do componente.	sim
SADL	Especifica os tipos de dados suportados	Escalonamento do processo usando prioridade estática	Cálculo Matemático	nenhum
Rapide	connection, inline	Informações de prioridade para análise de escalonabilidade	Consiste de um conjunto de regras de transição	sim
Dawin	binding, inline, nenhuma modelagem explícita de interações entre componentes	Informações de prioridade para análise de escalonabilidade	Usa CORBA	sim
UniCon	conector;	Informações de prioridade para análise de escalonabilidade	Atributos para análise de escalonabilidade	sim
AADL	conectores (portas, parâmetros, data access)	Security Level	usa subprograms C/C++, ADA; máquinas de estado	sim

Tabela 3 – Comparação entre ADLs
Fonte: fonte:(CHKOURI; BOZGA, 2009)

automotivo.

Considerando a escolha da linguagem AADL como a linguagem dos modelos alvo gerados pelo mecanismo de transformação de modelos proposto nesta tese de doutorado, a próxima seção apresenta em mais detalhes a linguagem AADL.

2.4 LINGUAGEM DE PROJETO E ANÁLISE DE ARQUITETURA - AADL

A Linguagem de Projeto e Análise de Arquitetura (AADL) padronizada pela SAE (*Society of Automotive Engineers*) em 2004 (P. Feiler and J. Hudak., 2006) foi desenvolvida para o domínio aviônico e é utilizada para modelar sistemas embarcados críticos e de tempo real. Ela define a arquitetura de um sistema como um conjunto de componentes interligados, e a modelagem dos componentes consiste em descrever suas interfaces, sua implementação e suas propriedades. A linguagem AADL possui representação textual, gráfica e XML (que facilita o intercâmbio de modelos entre ferramentas). Além disso, AADL é uma linguagem extensível, ou seja, linguagens externas podem ser utilizadas para definir anexos. AADL possui anexos para especificar o comportamento detalhado das aplicações, para a representação de dados e a modelagem de erros, bem como diretrizes para geração de código.

A representação gráfica da AADL proporciona uma visão de alto nível do sistema, ela facilita uma apresentação visual da hierarquia estrutural do sistema, da topologia de comunicação, e fornece uma base para perspectivas distintas da arquitetura. A representação textual permite visualizar e editar todos os detalhes do modelo de arquitetura do sistema. E a representação XML permite que o modelo AADL possa ser processado por ferramentas de terceiros, ou seja, as instâncias de um modelo de arquitetura do sistema são criadas e armazenadas como especificações (modelos) representados em XML, e as ferramentas de análise e de geração de código operam sobre estas instâncias do sistema. A Figura 9 resume as representações alternativas de uma especificação AADL, e a Figura 10 apresenta a notação gráfica dos componentes AADL.

Os conceitos fundamentais da linguagem e os elementos chave de uma especificação AADL são resumidos na figura 11. Em AADL, componentes são definidos através de declarações *Component Type* e *Component Implementation*. Uma declaração *Component Type* define a categoria e as interfaces (features) do componente. Uma declaração *Component Implementation* define a estrutura interna de um componente em termos de subcomponentes, conexões com subcomponentes, sequência de chamadas de *subprograms*, *modes*, e as implementações dos *flows* e *properties*. Os componentes AADL são agrupados em categorias de software, de plataforma de execução (hardware), composto e generic. Os *Packages* permitem a organização dos elementos AADL em grupos

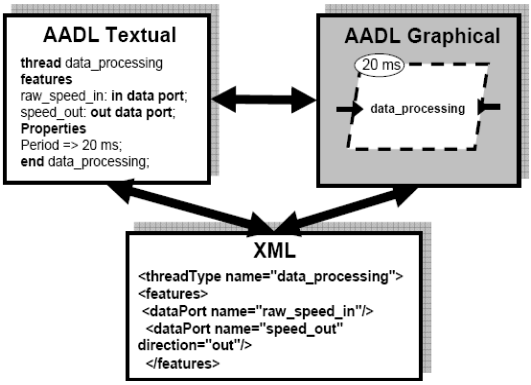

Figura 9 – Representações AADL
Fonte: (P. Feiler and J. Hudak., 2006)

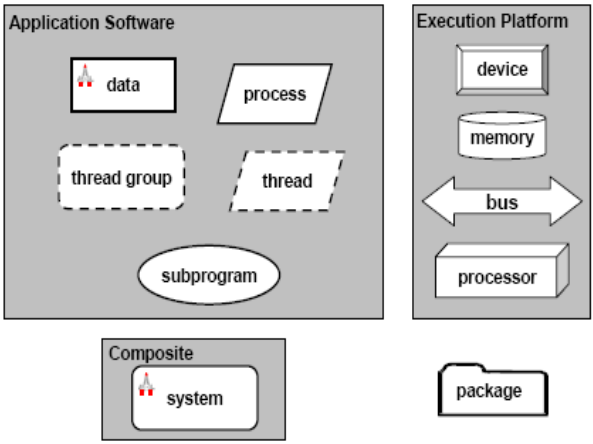

Figura 10 – Notação Gráfica dos componentes AADL
Fonte: (P. Feiler and J. Hudak., 2006)

nomeados. *Property Sets* e *Annnex Libraries* habilitam um projetista estender a linguagem e personalizar uma especificação AADL para atender a um projeto de domínio, ou necessidades específicas.

O padrão AADL define quatro conjuntos distintos de categorias de componentes para descrever um sistema. A Tabela 4 lista as categorias de componentes AADL existentes, seus respectivos componentes, e a combinação possível entre eles, ou seja, a relação de subcomponentes

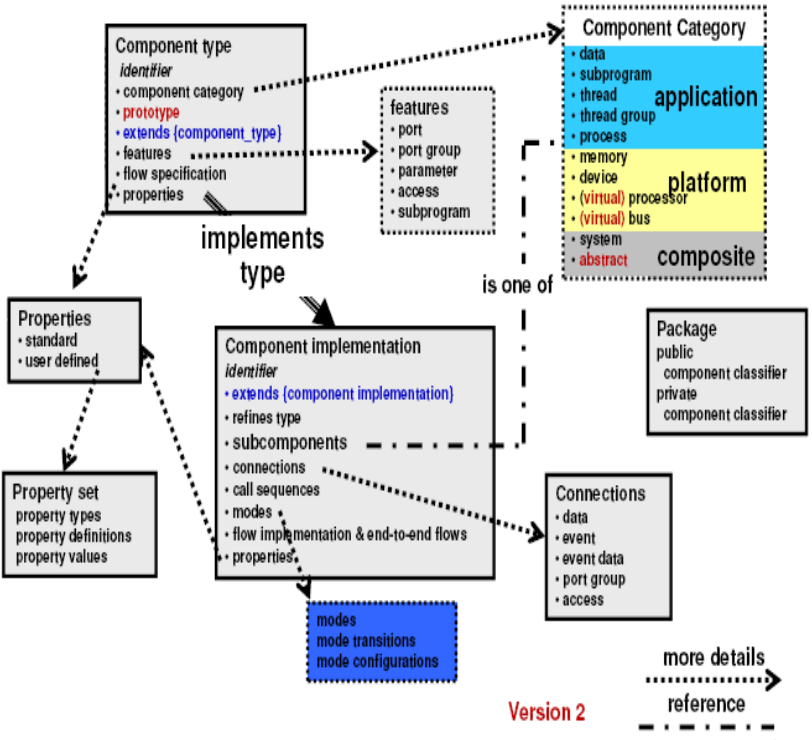

Figura 11 – Componentes e Conexões AADL
Fonte: (FEILER, 2010)

que um componente suporta.

Em relação a tabela 4:

- **Componentes do tipo software:** são utilizados para descrever a arquitetura de software de um sistema embarcado.

  thread: é uma unidade escalonável de execução simultânea;

  data: são tipos de dados e dados estáticos no código-fonte;

  thread group: é uma unidade de composição para a organização de threads;

  process: é um espaço de endereço protegido;

  subprogram: representa um código executável que pode ser chamado seqüencialmente.

Categoria de Grupo	**Categoria de Componente**	**Subcomponentes permitidos**	**Subcomponente permitido de**
Software	process	thread data thread group	system
	thread	data	process thread group
	data	data	process thread data threadgroup system
	threadgroup	data thread	process threadgroup
	subprogram	nenhum permitido	nenhum permitido
Plataforma de Execução	processor	memory	system
	memory	memory	processor memory system
	bus	Nenhum permitido	system
	device	Nenhum permitido	system
Composto/ Híbrido	system	process data processor memory bus device system	system
Genérico	abstract	nenhum permitido	nenhum permitido

Tabela 4 – Relação permitida entre componentes e subcomponentes AADL
Fonte: Adaptado a partir de (P. Feiler and J. Hudak., 2006)

- **Componentes do tipo Plataforma de Execução:** são utilizados para descrever a arquitetura de hardware do sistema embarcado.

  memory: são componentes que armazenam dados e código;

  bus: são componentes que permitem o acesso entre os componentes da plataforma de execução;

  virtual bus: representa uma abstração de comunicação, como um canal virtual ou um protocolo de comunicação. Componente adicionado na AADL v2;

  processor: são componentes que executam threads;

  virtual processor: recurso lógico que é capaz de escalonar

e executar threads que devem estar vinculadas a um ou mais processadores físicos. Componente adicionado na AADL v2;

device: são componentes que fazem interface e representam o ambiente externo.

- **Componente híbrido ou composto:** permitem a integração de outros componentes em unidades distintas dentro da arquitetura

  system: são composições que podem ser constituídas de outros sistemas, bem como de componentes de software ou de hardware.

- **Categoria Genérico:** especifica a intenção de gerar um componente. Esta categoria foi adicionada na AADL v2.

  abstract: define um componente neutro (conceitual) que pode ser refinado em outra categoria de componente. Componente adicionado na AADL v2.

Com a AADL é possível especificar modelos que descrevem individualmente o modelo de arquitetura de software e o modelo de arquitetura de hardware de um sistema, e também a integração entre estes modelos. Para que um modelo AADL possa ser considerado completo, os componentes de software devem ser mapeados sob uma plataforma de execução por meio de relações de ligação. Essas ligações definem onde o código é executado, e onde os dados e o código executável são armazenadas dentro de um sistema. Por exemplo, uma *thread* deve estar ligada a um processador para sua execução, e um processo deve estar vinculado a uma memória. Da mesma forma, as conexões entre os componentes dentro de um sistema devem estar ligados a componentes da plataforma de execução (barramentos). Uma ligação física entre um *device* e um processador também pode ser estabelecida por um barramento.

A semântica bem definida dos conceitos da AADL permite que vários tipos de verificações e análises quantitativas possam ser aplicadas às instâncias de um modelo AADL. Entretanto, antes de submeter a instância de um modelo AADL a análises e verificações, é necessário especificar algumas propriedades de execução das *threads*, tais como *dispatch_protocol*, *period*, *deadline* e *compute_exectuion_time*. Além das políticas de escalonamento e a velocidade do processador. Depois disso, e uma vez tendo estabelecido as ligações das *threads* com o processador, é possível avaliar a utilização do processador, e fazer uma análise de

escalonamento para determinar se as ligações existentes são aceitáveis. Além disso, também é possível atribuir orçamentos de recursos em termos de ciclos de CPU e memória para os subsistemas da aplicação, e capacidades de recursos para a plataforma de execução. Diante desses dados, é possível analisar diversas ligações entre os componentes da aplicação com a plataforma de execução, e garantir que os orçamentos não excedam a capacidades específicas da plataforma.

Outro recurso interessante da AADL é a capacidade que ela possui de modelar e analisar caminhos de fluxo de dados através de um sistema. Por exemplo, é possível analisar o tempo necessário para um sinal viajar a partir da unidade de interface, por meio do sistema de controle, até o acionamento do atuador, isto em resultado a uma ação específica. Para que isto seja possível, é preciso adicionar no modelo AADL especificações de fluxos nos componentes individuais do sistema (*processos e devices)*, e especificar o fluxo *ent-to-end* no sistema raiz do modelo AADL, e com base nestas especificações realizar análises de fluxo, como por exemplo, a análise de tempo de resposta *end-to-end*.

Um modelo AADL também pode ser submetido à verificação formal de propriedades, para assegurar a correção do comportamento do modelo. Porém, antes disso, é preciso especificar o comportamento dos *devices* e das *threads* utilizando o anexo comportamental da AADL. Dentre os pontos passíveis de verificação, destacam-se as propriedades comportamentais das *threads*, tais como: vivacidade, ausência de bloqueio e justiça. A cadeia de verificação disponível no projeto Topcased, por meio do plugin AAADL2FIACRE (BERTHOMIEU et al., 2010) torna possível que modelos descritos na linguagem AADL sejam transformados para modelos com um nível mais baixo de abstração na linguagem Fiacre, que, por sua vez, são transformados para modelos TTS, com um nível de abstração ainda mais baixo, servindo de entrada para a ferramenta de verificação SELT (BERTHOMIEU*; RIBET; VERNADAT, 2004). A Figura 12 apresenta a cadeia de verificação de propriedades comportamentais do projeto Topcased.

Além das verificações e análises já comentadas, o modelo AADL também pode ser submetido a outros tipos de análise, como a análise de erros, a qual faz uso do anexo de erros da AADL.

As ferramentas listadas a seguir oferecem suporte a criação e edição de modelos AADL:

- **OSATE:** O Instituto de Engenharia de Software da Universidade Carnegie Mellon desenvolveu o *Open Source AADL Tool Environment*(OSATE). OSATE é composto por um conjunto de *plugins* para a plataforma Eclipse. O OSATE possui um compilador

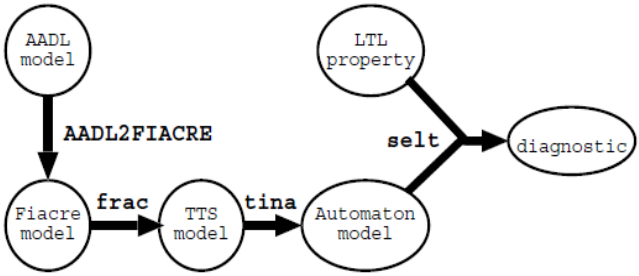

Figura 12 – Cadeia de Verificação do Projeto Topcased
Fonte:(CORREA et al., 2010)

para AADL textual, um editor gráfico para AADL, um gerador de instâncias de modelos AADL, e suporte para o formato de intercâmbio de dados XMI baseado em XML para AADL, com base em sua especificação do metamodelo. O editor de gráfico Adele para OSATE foi fornecido pela Ellidiss (www.ellidiss.com). Uma série de ferramentas de análise e geração de código fonte foram integrados com OSATE através do formato de intercâmbio XMI. OSATE está integrado com a suíte de ferramentas do TOPCASED para o desenvolvimento de sistemas embarcados.

- **STOOD:** É uma ferramenta comercial para o desenvolvimento de sistemas embarcados (www.ellidiss.com), que abrange o ciclo de vida completo partindo de requisitos até a construção do sistema. Originalmente, desenvolvido para apoiar HOOD-RT, foi estendido para suportar modelos AADL. Ele fornece uma interface gráfica para o projeto arquitetônico e gera AADL textual. Ele inclui um AADL Inspector para validação do modelo AADL e uma capacidade para extração de informações arquiteturais a partir do código fonte. STOOD foi integrado com ferramentas de análise e simuladores como interface com outras cadeias ferramenta AADL, incluindo OSATE e Ocarina, para tirar proveito de sua capacidade de análise e geração.

- O conjunto de ferramentas da Rockwell Collins META estende o OSATE, fornecendo um editor gráfico para AADL baseado em um editor SysML, uma ferramenta padrão de projeto arquitetônico, uma ferramenta de verificação do modelo estático, e uma ferramenta de verificação de composição. O editor gráfico

foi interligado com OSATE para converter os modelos textuais AADL editados dentro OSATE e modelos gráficos editados dentro do Enterprise Architect, um editor SysML.

- **TASTE:** O TASTE (*ASSERT Set of Tools for Engineering*) é um conjunto de ferramentas que foi desenvolvido pela Agência Espacial Europeia (ESA) e seus parceiros. Ele fornece um conjunto de editores gráficos para criar a arquitetura de software, a arquitetura de hardware, e a implantação do software no hardware. Estes editores fornecem uma interface de usuário simplificada, eliminando a necessidade de aprender a sintaxe textual de AADL.

- Ferramentas UML usando um perfil para AADL também podem criar modelos AADL. Uma implementação do protótipo de tal perfil foi feito para Rhapsody na iniciativa chamada SAVI (*System Architecture Virtual Integration*) da indústria aeroespacial.

- **AADL Inspector**: É uma ferramenta independente para verificar a consistência de modelos AADL textual (www.ellidiss.com). É integrado com a ferramenta de análise de escalonamento Cheddar e com um motor de simulação multiagentes para a simulação dinâmica de modelos AADL.

- **TOPCASED**: É um ambiente *opensource* para desenvolvimento de sistemas embarcados críticos baseado na plataforma do projeto de modelagem eclipse, ele fornece um conjunto de ferramentas tais como: editores gráficos e de texto, tradutores de modelos, geradores de teste, de documentação e de código, ferramentas de controle de versão, ferramentas de verificação de modelo, ferramentas de validação de requisitos, entre outros recursos, os quais permitem lidar com os diversos modelos que podem compor o projeto de um sistema embarcado crítico, incluindo modelos AADL (FARAIL et al., 2006).

- Outras organizações têm criado ferramentas que geram modelos AADL a partir do conteúdo de bases de dados de projetos, e para extrair modelos AADL a partir de modelos Simulink existentes. A abordagens que apresentaram resultados mais significativos na extração de modelos AADL a partir de modelos Simulink, são apresentadas em mais detalhes na próxima seção.

2.5 CONSIDERAÇÕES

Embora o mecanismo de suporte à execução da transição de projeto entre a modelagem funcional e a modelagem arquitetural durante o processo de desenvolvimento de CPSs proposto por esta tese de doutorado seja genérico, ou seja, que o mecanismo proposto possa ser utilizado com diferentes combinações de ferramentas de modelagem funcional e de linguagem de descrição de arquitetura, foi necessário definir uma ferramenta de modelagem funcional e uma linguagem de descrição de arquitetura para gerar, respectivamente, o modelo fonte e o modelo destino, e assim tornar possível a avaliação e a validação do processo de transformação de modelos por este trabalho de pesquisa.

A escolha da ferramenta Simulink como ferramenta de modelagem funcional para gerar os modelos fonte do mecanismo de transformação de modelos proposto nesta tese de doutorado, foi devido ao seu grau de maturidade e popularidade entre os engenheiros de controle e automação. Além disso, a ferramenta Simulink possui um conjunto de características comum às outras ferramentas de modelagem funcional apresentadas neste capítulo, como o fato de possibilitar a organização de seus modelos de forma hierárquica utilizando o conceito de diagrama de blocos.

A escolha da linguagem de descrição de arquitetura dos modelos alvo gerados pelo mecanismo de transformação de modelos proposto foi feita levando em consideração o resultado das comparações feitas entre as ADLs apresentado anteriormente neste capítulo. Portanto, a escolha da linguagem AADL deve-se ao fato dela fornecer recursos que possibilitam descrever individualmente o modelo de arquitetura de software, o modelo de arquitetura de hardware, composto por processadores, barramentos e memória, além de possibilitar especificar a integração entre estes modelos para compor o modelo de arquitetura completo do sistema embarcado. Outra questão que influenciou sua escolha foi o fato da quantidade crescente de ferramentas de análise disponíveis para esta linguagem, o que aponta que, tanto a comunidade científica quanto a indústria estão usando a linguagem AADL para explorar novas técnicas de análise para o modelo de arquitetura de CPS (DELANGE et al., 2010). Por fim, destaca-se que as abordagens existentes voltadas à integração de modelos funcional e de arquitetura concentram-se na integração de modelos funcionais Simulink ou Scade com modelos de arquitetura de sistemas especificados usando a linguagem de descrição de arquitetura AADL.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta as abordagens existentes voltadas ao mapeamento e a integração de modelos funcionais Simulink com modelos de arquiteura AADL durante a concepção de CPSs.

3.1 PROJETO ASSERT

De acordo (ESTEC-ESA, 2008), o projeto ASSERT propõe uma metodologia de desenvolvimento de software que possibilita projetar sistemas complexos usando várias ferramentas de modelagem através de uma cadeia de ferramentas. O projeto ASSERT foi financiado pela ESA (Agência Espacial Européia) e pela Comissão Européia, com o propósito melhorar o processo de desenvolvimento de sistemas embarcados críticos de tempo real.

A metodologia ASSERT é apoiada por um conjunto de ferramentas desenvolvidas ao longo do projeto. O principal objetivo da cadeia de ferramentas do projeto ASSERT é permitir que equipes de desenvolvimento de componentes individuais possam escolher a ferramenta de modelagem que melhor atenda aos requisitos do respectivo componente, e ao mesmo tempo assegurar a integração destes componentes com uma arquitetura global, isso, sem a necessidade de implementar código manualmente.

Para lidar com a diferença entre as notações dinâmicas e estáticas, a cadeia de ferramentas ASSERT utiliza um modelo de alto nível do sistema global, que descreve os subsistemas individuais e suas interfaces de forma abstrata. Isto foi possível através da utilização de uma ferramenta de modelagem independente e pela combinação das linguagens AADL e ASN.1 para fornecer uma descrição formal, precisa e completa da arquitetura do sistema e dos dados. Sendo assim, a linguagem AADL na sua forma gráfica é usada para descrever o modelo do sistema de alto nível, ou seja, os subsistemas, suas propriedades não-funcionais (por exemplo, WCET) e suas interfaces. Ela também é usada para descrever as configurações de implantação do sistema (por exemplo, qual subsistema é atribuído à qual CPU). A linguagem ASN.1 (*Abstract Syntax Notation number One*) é usada para descrever as mensagens trocadas entre os subsistemas. Em essência, a cadeia de ferramentas gerada no projeto ASSERT permite que códigos gerados por ferramentas de modelagem diferentes possam interagir de

forma transparente, sem a participação do usuário no processo de empacotamento de mensagens.

3.2 IME - *INTEGRATED MODELING ENVIRONMENT*

A empresa Emmeskay oferece serviços e ferramentas de suporte ao desenvolvimento baseado em modelos de sistemas embarcados. Uma das ferramentas oferecidas por tal empresa é o IME (*Integrated Modeling Environment*) que utiliza como padrão a Linguagem Descrição da Arquitetura de Sistemas (SADL)(IME, 2011).

Os estudos apresentados em (RAGHAV et al., 2009b), (RAGHAV et al., 2009a) e (RAGHAV; GOPALSWAMY, 2009), demonstram a possibilidade do IME oferecer suporte para gerar um modelo de arquitetura do sistema em AADL a partir de informações estruturais extraídas de um modelo funcional Simulink; e também demonstram a possibilidade do IME permitir anexar submodelos Simulink aos componentes de um modelo de arquitetura de sistema em AADL pré-existente. Entretanto, para que isso seja possível, um modelo de arquitetura do sistema intermediário em *SADL* precisa ser marcado com marcas que representam conceitos AADL, para então ser transformado em um modelo AADL textual. Nesta abordagem, as marcas são chamadas de atributos. As marcas que são utilizadas para marcar um modelo em SADL, e que guiam as regras de transformação entre os modelos SADL e AADL são apresentadas na Tabela 5. Os mesmos estudos também apresentam a possibilidade de que um modelo de arquitetura descrito em AADL poder ser importado pelo IME, desde que o mesmo esteja em um formato XML (arquivo *.aaxl*). Neste caso, o mapeamento ocorre no sentido do modelo AADL para o modelo *SADL*. A Figura 13 demonstra o mapeamento suportado pelo IME entre os modelos Simulink, SADL e AADL

Modelo SADL	**Tipo de Componente AADL e Implementation**
Sem Atributos	System
Tipo IME com atributo Processor	Processor
Tipo IME com atributo Process	Process
Tipo IME com atributo Thread	Thread
Tipo Simulink	Subprogram

Tabela 5 – Conjunto de Marcas do IME
Fonte:(RAGHAV; GOPALSWAMY, 2009)

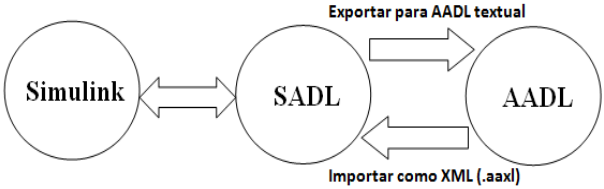

Figura 13 – Mapeamento Simulink-SADL-AADL
fonte:(RAGHAV; GOPALSWAMY, 2009).

3.3 POLYCHRONY

Polychrony é um conjunto de ferramentas de projeto auxiliado por computador que implementa o modelo de computação e comunicação para capturar semanticamente arquiteturas embarcadas denominadas GALS (Globalmente Assíncronas e Localmente Síncronas) (MA, 2010). Ele fornece uma representação deste modelo de computação através de um ambiente Eclipse para facilitar a sua utilização e interoperação com a heterogeneidade de linguagens e modelos comumente utilizados nos domínios de aplicação aeroespacial e automotivo. Baseado na linguagem Signal, o núcleo do Polychrony fornece uma ampla gama de serviços de análise, transformação, verificação e síntese para auxiliar o engenheiro com as tarefas necessárias que levam à simulação, teste, verificação e de geração de código para arquiteturas de software. Em dezembro de 2010, o conjunto de ferramentas Polychrony ficou disponível sob licença EPL e GPL v2.0 pelo INRIA (INRIA, 2010). Signal é uma linguagem para especificação e programação para aplicações embarcadas críticas e de tempo-real (LEGUERNIC et al., 1991). Polychrony é o resultado de pesquisas da equipe de projeto ESPRESSO (INRIA, 2010).

(YU et al., 2011) propõe uma abordagem para a questão da composição, integração e simulação de modelos heterogêneos em um fluxo de co-design de sistema utilizando Polycrhrony no contexto do projeto CESAR (*Cost-efficient methods and processes for safety relevant embedded systems*)(CESAR..., 2010). Neste caso, a ferramenta Polychrony foi integrada à *Reference Technology Platform* (RTP) do projeto CESAR para servir como um framework de cooperação para a modelagem e exploração de arquitetura. De acordo com a Figura 14, o comportamento do sistema é modelado usando o modelo de computação síncrono do Simulink/GeneAuto (TOOM et al., 2008), e o

modelo de arquitetura é especificado usando o modelo de computação assíncrono da AADL. Estes modelos de alto nível são transformados em programas Signal através do SME (Signal-Meta Eclipse). Signal suporta um modelo de computação síncrono multi-clock que permite a descrição de sistemas localmente síncronos e globalmente assíncronos. Posteriormente o código C ou Java é gerado a partir dos programas Signal. A simulação pode, por exemplo, ser realizada com a finalidade de avaliação de desempenho. O Polychrony e suas ferramentas associadas permitem uma simulação rápida e eficaz, e uma avaliação e validação de arquiteturas a nível de sistema, sem a necessidade de traduções adicionais para outros formalismos, ou seja, a verificação formal, a simulação, e a análise podem ser realizadas diretamente neste formalismo (signal), sem a necessidade de traduções complementares para outros formalismos (YU et al., 2011).

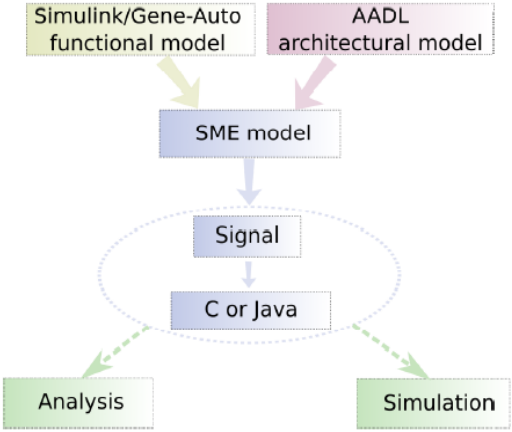

Figura 14 – Processo de Design para a Modelagem e Simulação Heterogênea dentro do Polychrony
fonte:(YU et al., 2011)

3.4 INTEGRAÇÃO DOS CÓDIGOS-FONTE DE MODELOS SIMULINK E AADL

O objetivo principal da proposta apresentada em (DELANGE et al., 2010) é demonstrar como integrar o código fonte gerado a partir de modelos Simulink com o código fonte gerado a partir de modelos AADL,

tendo como objetivo final a obtenção do código executável completo do sistema projetado, possibilitando o desenvolvimento de todo o sistema a partir de modelos.

A solução de integração de modelos proposta nesta abordagem possibilita que dois fluxos de trabalho possam ser executados. O primeiro fluxo de trabalho define que inicialmente devem ser gerados os modelos funcionais Simulink, e, posteriormente, deve ser gerado um modelo de arquitetura de software descrito em AADL. Em seguida, deve ser feita a associação dos modelos funcionais Simulink com os componentes de software do modelo de arquitetura AADL. Por último, o modelo de arquitetura deve ser anotado e submetido a ferramentas de verificação e análises disponíveis para modelos AADL. O segundo fluxo de trabalho inicia com o projeto da arquitetura de software descrito em AADL, em seguida os componentes funcionais são implementados em Simulink, sendo que suas interfaces devem coincidir com a interface descrita no modelo AADL. Em seguida, os componentes de software do modelo de arquitetura AADL são refinados para fazer referência a estes componentes funcionais.

Nesta abordagem, a integração dos modelos subdivide-se em três atividades: (i) Importação dos dados do aplicativo para AADL; (ii) Importação das funções do aplicativo para AADL; e (iii) Geração e integração de código.

A importação dos dados específicos do modelo Simulink é viabilizada através do componente AADL *data*. A importação das funções do aplicativo é feita através do componente AADL *subprogram*, o qual modela o fluxo de instruções executadas em outro componente AADL chamado *thread*. A geração de código em nível de aplicação é de responsabilidade dos geradores de código específicos, neste caso, o *Real-Time Workshop* para Simulink e o *KCG* para SCADE, os quais fornecem funcionalidades importantes que podem ser usadas para conseguir o intercâmbio de dados entre os níveis de aplicação e de arquitetura. O código fonte a nível de arquitetura é gerado pelo conjunto de ferramentas OCARINA (LASNIER et al., 2009).

3.5 *PLUGINS IMPORTER.SIMULINK E IMPORTER.SCADE* PARA OSATE2

O *Software Engineering Institute* (SEI) está implementando um importador de modelos Simulink (em formato .slx) e um importador de modelos Scade para o OSATE2, que visa permitir ao ambiente OSATE2

importar um modelo Simulink no formato SLX ou um modelo Scade e gerar a partir deste o esqueleto do modelo de arquitetura de um sistema em AADL. Este é um projeto em desenvolvimento, ele foi apresentado na *SAE AeroTech Conference & Exhibition* que ocorreu na cidade Montreal, Quebec, Canada, em setembro de 2013. Uma versão de teste dos *plugins importer.simulink* e *importer.scade* foi disponibilizada no site *https://github.com/osate/ osate2-plugins/tree/develop* em fevereiro de 2014.

O projeto do *plugin importer.simulink* encontra-se em uma fase de desenvolvimento precoce, e por este motivo, até o momento, não existe qualquer publicação, documentação ou relatório técnico disponível para consulta. O importador em questão está sendo desenvolvido para o programa SAVI (*The System Architecture Virtual Integration Program*).

O programa AVSI SAVI ("savvy") ((AVSI), 2010) é uma colaboração entre os interessados no desenvolvimento de sistemas aeroespaciais, e tem como objetivo avançar o estado da arte das tecnologias que permitem integração virtual de sistemas complexos. Membros atuais do Programa de SAVI incluem Airbus, Boeing, BAE Systems, U.S. DoD, Embraer, U.S FAA, Goodrich, Honeywell, U.S. NASA, Rockwell Collins e o Software Engineering Institute/CMU. O SAVI é um projeto dentro do Aerospace Vehicle Systems Institute, uma cooperativa de pesquisa da indústria aeroespacial, que faz parte do Texas A&M University System, e cujos membros executam projetos colaborativos de pesquisa e tecnologia aplicada.

3.6 LIMITAÇÕES DAS ABORDAGENS EXISTENTES

Iniciando pelo projeto ASSERT, é possível reconhecer que ele trouxe contribuições significativas no domínio de desenvolvimento de sistemas embarcados críticos, apresentando uma metodologia de desenvolvimento bem estruturada e um conjunto de ferramentas que oferecem suporte à aplicação de tal metodologia, possibilitando a utilização de várias ferramentas de modelagem e, consequentemente, diferentes modelos durante o processo de desenvolvimento e um sistema embarcado crítico. Entretanto, acredita-se não ser uma boa prática o modelo proposto pelo ASSERT de iniciar um projeto pela definição da arquitetura de software. Os modelos funcionais podem ser criados por diferentes ferramentas de modelagem para atender tal arquitetura de software projetada previamente. Desta forma, o modelo de arquitetura

de software não é gerado a partir de um modelo funcional existente, mas sim modelos funcionais específicos é que são criados para serem consistentes de uma arquitetura de software previamente especificada.

Já a proposta de adaptação da ferramenta IME possui algumas limitações quanto ao mapeamento de modelos SADL para modelos AADL. Isto pode ser atribuído a falta de compatibilidade completa entre as duas DSLs. De acordo com (RAGHAV; GOPALSWAMY, 2009), as portas de dados SADL são traduzidas para portas de dados AADL, já o mapeamento de outras portas de dados mais complexos não é suportado atualmente. Além disso, a exportação para AADL de informações adicionais acrescentadas pelo usuário diretamente no IME é limitada. Outra limitação, é que o IME não oferece suporte a todos os componentes de hardware da AADL. Esta última limitação, em particular, pode prejudicar a criação de um modelo de arquitetura do sistema detalhado e mais próximo do produto final. O IME também não extrai dos modelos funcionais Simulink possíveis modos de operação ou comportamento dos componentes de software, uma vez que ele não suporta mapear o comportamento de um componente funcional que esteja representado por um diagrama Stateflow. Isso acontece porque a linguagem SADL não suporta representar comportamento usando *state-machines* (CHKOURI; BOZGA, 2009), veja Tabela 3. Além disso, o estudo viabilizando a integração de modelos Simulink e AADL proposto pela Emmeskay através da ferramenta IME, foi feito em caráter experimental, e parece que não está sendo comercializado atualmente como parte da ferramenta.

Em (YU et al., 2011), a ferramenta Polychrony foi integrada ao projeto CESAR para servir como um framework de cooperação para modelagem e exploração de arquitetura. No escopo do projeto CESAR a ferramenta de co-modelagem Polychrony suporta a importação de modelos de alto nível Simulink (funcional) e especificações AADL (arquitetural). Esta função de importação está atualmente implementada por duas transformações diferentes, a saber, Simulink-to-Signal e AADL-to-Signal. Para integrar os programas Signal resultantes dessas transformações, algumas interfaces Signal precisam ser executadas manualmente. A composição de modelos Simulink e AADL, depende, portanto, de projetistas de sistemas para implementar estas interfaces, o que torna difícil a manutenção e a validação do modelo resultante.

Em relação a abordagem apresentada em (DELANGE et al., 2010). Considerando que o estudo descrito foi desenvolvido como uma prova de conceito, focando exclusivamente a integração dos códigos-fonte gerados a partir de modelos Simulink e AADL para compor o código-

fonte completo da aplicação, algumas questões importantes não foram contempladas. Dentre elas: a) Os modelos Simulink e AADL são criados de forma independente, ou seja, um não é gerado a partir do outro; b) Os componentes dos modelos são interligados por associações feitas de forma manual; c) Não garante a sincronização dos modelos, ou seja, possíveis alterações em um determinado modelo não são necessariamente refletidas no outro; e d) A consistência entre a interface dos blocos de um modelo Simulink e a interface dos componentes de software AADL deve ser garantida de forma manual.

Quanto ao *plugin importer.simulink*, aparentemente ele não suporta a importação de modelos Simulink em formato *mdl*, o mapeamento estrutural gera apenas componentes do tipo *system*, e o mapeamento comportamental suporta o mapeamento de blocos *Stateflow* hierárquicos. No entanto, o foco do mapeamento comportamental parece ser a identificação de problemas arquitetônicos relacionados ao compartilhamento de dados, e não a verificação de propriedades comportamentais. O *plugin importer.simulink* parece também não suportar o mapeamento de possíveis modos de operação do sistema especificados em modelos funcionais Simulink para o modelo de arquitetura AADL correspondente.

A Tabela 6 apresenta um quadro comparativo com algumas perguntas-chave que permitem a identificação das principais características e limitações das abordagens de transformação e integração de modelos Simulink e AADL apresentadas anteriormente.

	ASSERT	**IME**	**Polychrony**	**Integração Códigos-Fonte**	**Plugin importer. simulink**
Gera Modelo de Arquitetura a partir de Modelo Funcional?	não	sim	não	não	sim
Prevê um mapeamento direto entre modelos Simulink e AADL?	não	não	não	não	sim
Aplica a técnica de transformação de modelos M2M?	sim	sim	sim	não	sim
Suporta mapeamento estrutural?	sim	sim	não	não	sim
Suporta mapeamento comportamental?	não	não	não	não	sim
Suporta mapeamento de modos de operação?	não	não	não	não	não
Possui suporte de ferramenta computacional?	sim	sim	sim	não	sim

Tabela 6 – Quadro Comparativo dos Trabalhos Relacionados

3.7 CONSIDERAÇÕES

Tanto quanto foi possível verificar, não há uma solução definitiva para o problema do mapeamento de modelos Simulink para modelos AADL. Algumas das abordagens existentes fornecem maneiras de relacionar os modelos, mas não uma alternativa no sentido de uma transformação de modelos. Dentre os trabalhos ((ESTEC-ESA, 2008; RAGHAV et al., 2009b; YU et al., 2011)) que cobrem mapeamento, nenhum destes aborda o mapeamento comportamental. Em relação ao mapeamento estrutural, todos os estes fazem uso de uma linguagem intermediária, limitando as possibilidades de mapeamento e a posterior manutenção da integração dos modelos.

O *plugin importer.simulink* e o mecanismo de transformação de modelos apresentado neste trabalho de pesquisa parecem ter o mesmo objetivo, apesar de aparentemente adotarem uma abordagem diferente para realizar os mapeamentos estrutural e comportamental entre os modelos Simulink e AADL. O projeto do *plugin importer.simulink* está em um estágio inicial de desenvolvimento se comparado com o projeto do mecanismo de transformação de modelos apresentado neste documento de tese. Na medida do que foi possível compreender, em consulta ao pouco material encontrado e a versão de teste disponível do *plugin importer.simulink*, ele não suporta a importação de modelos Simulink em formato mdl, o mapeamento estrutural gera apenas tem componentes do tipo *system* e mapeamento comportamental suporta o mapeamento de blocos *Stateflow* hierárquicos. No entanto, o foco do mapeamento comportamental parece ser a identificação de problemas arquitetônicos relacionados ao compartilhamento de dados, e não a verificação de propriedades comportamentais como é o caso do mecanismo proposto por esta tese de doutorado. Além disso, parece que o *plugin importer.simulink* não suporta o mapeamento de possíveis modos de operação do sistema especificados em modelos funcionais Simulink para o modelo de arquitetura AADL correspondente.

Na realidade, foi muito difícil reproduzir os resultados dos trabalhos relacionados. Isso se deve ao fato de que a documentação disponível é na maioria das vezes incompleta, e também devido a falta de ferramentas computacionais de suporte disponíveis para auxiliar a aplicação das abordagens propostas. Todas estas questões motivaram a apresentação de um mecanismo capaz de oferecer suporte ao mapeamento de aspectos estruturais, comportamentais e de possíveis modos de operação de um modelo funcional Simulink para um modelo preliminar de arquitetura AADL durante o processo de desenvolvimento de CPSs.

4 TRANSFORMAÇÃO ASSISTIDA DE MODELOS - AST

Este capítulo apresenta o ponto central desta tese de doutorado, o qual consiste de um mecanismo que contribui com a geração da arquitetura de Sistemas Computacionais Embarcados (ECS), mais especificamente, um mecanismo para transformar um modelo funcional em um modelo de arquitetura de software preliminar durante o processo de desenvolvimento de CPSs. Tal mecanismo é chamado de Transformação Assistida de Modelos (AST).

4.1 CONTEXTUALIZAÇÃO DO MECANISMO PROPOSTO

O mecanismo de Transformação "ASsistida" de modelos (AST) tem por finalidade contribuir com a redução do grau de dificuldade de relacionar os componentes de um modelo funcional com os componentes do modelo de arquitetura de software do respectivo CPS em desenvolvimento.

A Figura 15 apresenta uma re-leitura do processo de desenvolvimento proposto por (CORREA et al., 2010) feita por esta autora e seus orientadores. O processo em questão adota diferentes linguagens de modelagem ao longo do ciclo de vida do desenvolvimento, de modo que artefatos com fins específicos podem ser gerados ao longo do projeto. A necessidade de adotar mais do que uma linguagem está relacionada ao fato de que há uma enorme lacuna semântica entre as necessidades dos usuários e o código final que precisa ser escrito para implementá-las. A introdução dessas linguagens permite que a equipe de desenvolvimento mude gradativamente a representação do sistema, partindo de uma linguagem mais abstrata (ou informal) para outra representação mais concreta (ou rigorosa). Isso também permite a adição gradual de detalhes relevantes nos modelos.

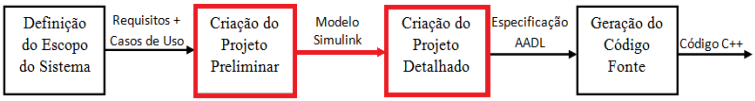

Figura 15 – Principais atividades e artefatos do método para desenvolver CPSs.

As linguagens utilizadas neste processo permitem que os seguin-

tes artefatos façam parte do processo de desenvolvimento, para apoiar em diferentes níveis a especificação, o projeto e a implementação do sistema em questão:

1. Especificação de Requisitos (escrito em uma linguagem natural)
2. Casos de Uso e suas descrições textuais
3. Modelos de Simulação
4. Modelos de Arquitetura Software/Hardware
5. Programas de Software

É compreensível que, para um projeto diferente, para um tipo distinto de sistema, ou para um contexto diferente, possam ser adotadas outras linguagens. Além disso, deve ser óbvio de que é possível escolher uma linguagem diferente para fins de implementação.

Em relação ao encadeamento das atividades apresentado na figura 15:

- *Definição do Escopo do Sistema*: o primeiro passo do processo de desenvolvimento é definir o escopo do sistema. Isto implica em elicitar os requisitos das fontes (normalmente os stakeholders). Este é um passo não trivial a ser realizado, uma vez que implica na identificação das necessidades dos usuários no que diz respeito ao sistema a ser construído. Há muitas técnicas que podem ser usadas nesta etapa, como entrevistas, análise de tarefas, análise de domínio, a introspecção, o debate e a observação. Um bom estudo sobre técnicas de elicitação pode ser encontrado em (ZOWGHI; COULIN, 2005). Como é habitual nestas situações, não há regras rígidas para decidir quais técnicas utilizar. As técnicas que são utilizadas para uma dada situação dependem de uma avaliação criteriosa feita pelo engenheiro. O resultado final desta etapa é uma especificação dos requisitos dos usuários, tanto funcionais quanto não funcionais, escrita em uma linguagem natural. Esta especificação serve principalmente como meio de comunicação entre os usuários e os membros do projeto.

  Quando a especificação de requisitos é considerada completa, a equipe de desenvolvimento pode produzir um diagrama de casos de uso. Ao construir este diagrama, a equipe de desenvolvimento precisa definir duas questões principais:

1. Os atores que o sistema interage. Isto é, o ambiente ou contexto do sistema;
2. Os casos de uso fornecidos pelo sistema, ou seja, as principais funcionalidades disponíveis para seus usuários.

- *Criação do Projeto Preliminar*: O próximo passo consiste na criação de um projeto preliminar para o CPS, com foco tanto na representação matemática do dispositivo eletromecânico a ser controlado quanto na criação de um primeiro esboço de sua solução arquitetônica. Devem participar da execução desta tarefa membros da equipe com conhecimento em sistemas de controle e engenheiros da computação.

  Nesta atividade, é sugerido que a equipe de desenvolvimento utilize uma linguagem que permita a estruturação do CPS em componentes e/ou subsistemas, e que também permita que o CPS possa ser simulado - tanto o controlador, quanto o ambiente. A linguagem precisa ser executável (para fins de simulação) e gráfica, permitindo que o designer represente visualmente a arquitetura do CPS. Além disso, a equipe de desenvolvimento também deve julgar sobre a necessidade da utilização de modos de operação para o sistema que esta sendo projetado.

  Cada componente da arquitetura deve ser refinado, introduzindo novos componentes ou a descrição do comportamento dinâmico do componente, ou ambos. Este processo é repetido até que todos os componentes estejam adequadamente especificados.

- *Criação do Projeto Detalhado*: A criação de um projeto detalhado é obrigatório em qualquer projeto de engenharia, e não é diferente ao projetar um CPS. Nesta etapa, a equipe do projeto deve produzir um modelo com um nível de detalhamento que torne sua implementação a mais simples possível. É ainda melhor se esse modelo puder ser objeto de análise, possibilitando que a equipe do projeto possa utilizar tal modelo para julgar decisões de projeto tomadas antes da implementação do modelo, de modo que possíveis erros de projeto possam ser antecipados e devidamente corrigidos.

  Especificamente no caso de um projeto CPS, a equipe deve decidir sobre a alocação das funcionalidades em processos e threads, e também sobre a implementação de tais tarefas em uma plataforma de execução (hardware), que por sinal também deve ser definida. Além disso, as diferentes restrições relacionadas à implementação

do sistema devem ser levadas em consideração, como por exemplo, restrições de tempo e de consumo de energia.

- *Geração do Código Fonte*: Assim que o projeto detalhado é concluído, é necessário traduzi-lo em uma linguagem de programação. Esta etapa também pode estar relacionada com o co-projeto de software/hardware, mas, por uma questão de simplicidade, é considerado apenas como geração de software. A geração de código pode ser executada manualmente ou automaticamente. Por conveniência, a geração automática de código é preferível.

Conforme já comentado no capítulo 1 deste documento, um dos pontos em aberto neste processo de desenvolvimento, é, justamente, a definição da maneira como se dá a transformação do projeto preliminar (modelo funcional Simulink) para o projeto detalhado do sistema (especificação AADL). Neste contexto, a AST é apresentada como um mecanismo para auxiliar a criação do modelo de arquitetura de software a partir de informações estruturais, operacionais (modos de operação) e comportamentais extraídas de um modelo funcional previamente especificado.

4.2 APRESENTAÇÃO DO MECANISMO PROPOSTO

Durante o processo de desenvolvimento de um CPS é comum a utilização de diferentes tipos de modelos. Dentre eles, é comum a existência do modelo da planta física do sistema, o qual apresenta a modelagem matemática do dispositivo eletromecânico a ser controlado, a existência do modelo funcional, que apresenta a especificação dos componentes de software do sistema em desenvolvimento. Este modelo em particular possibilita a execução de simulações dos algoritmos de controle, e em última instância ele fornece o código-fonte das funções do sistema. Já o modelo de arquitetura do sistema é responsável pela estruturação do software em termos de processos e *threads* e apresentando as propriedades de execução destes componentes do sistema. No modelo de arquitetura também é definida a plataforma de execução do sistema, em termos de processadores, memórias e barramentos, bem como suas propriedades de execução. O que torna possível a validação das características esperadas do sistema em termos de escalonamento, segurança e comportamento, isto, por meio da execução de diferentes tipos de análises e verificações.

A Figura 16 representa graficamente a relação entre os modelos

funcionais e o modelo de arquitetura de um CPS.

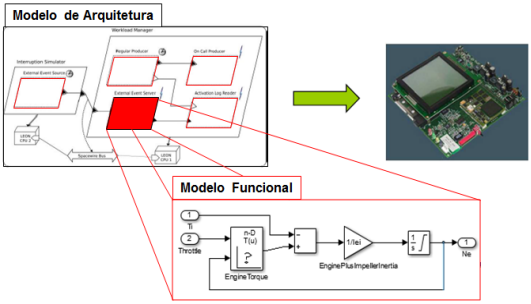

Figura 16 – Relação entre Modelos Funcionais e o Modelo de Arquitetura de um CPS

Como pode ser observado na Figura 16, os modelos funcionais e de arquitetura gerados durante o processo de desenvolvimento de um CPS são considerados complementares, uma vez que os modelos funcionais fornecem aos componentes de software do modelo de arquitetura os códigos-fonte das funcionalidades do sistema. Sendo assim, é fundamental que estes modelos sejam consistentes entre si, e que além disso, eles estejam devidamente integrados. Os códigos-fonte das funcionalidades do sistema podem ser gerados a partir de diagramas de blocos, o quais são frequentemente utilizados por engenheiros para especificar os algoritmos de controle de CPSs.

O processo de desenvolvimento de CPSs tende a ser uma atividade multidisciplinar. Como pode ser observado na Figura 17, enquanto os engenheiros de controle e automação estão preocupados com a modelagem funcional do sistema (implementação dos algoritmos de controle), os engenheiros de computação definem a arquitetura do sistema. Entretanto, um problema frequentemente encontrado neste cenário é a falta de integração e a inconsistência entre os diferentes modelos funcionais e de arquitetura do sistema.

Visando contornar tal problema e tornar a transição de projeto entre a modelagem funcional e a modelagem de arquitetura do sistema mais suave, esta tese de doutorado apresenta um mecanismo de transformação assistida de modelos, denominado AST, que sugere uma forma coerente de relacionar os elementos do modelo funcional com os componentes de software do modelo de arquitetura do sistema. A AST oferece suporte para a extração de informações de um modelo funcional, gerado em uma etapa anterior do processo de desenvolvimento, sem

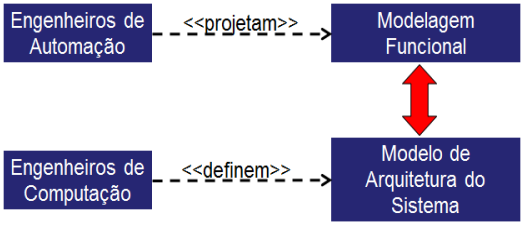

Figura 17 – Característica Multidisciplinar do Processo de Desenvolvimento de CPSs

que para isso seja necessário dominar técnicas de controle, nem tão pouco analisar o código-fonte gerado pelas ferramentas de modelagem e simulação. Para tanto, foram empregadas técnicas da MDE para estruturar um motor de transformação capaz de executar a transformação de modelos funcionais utilizados para a modelagem funcional e simulação em modelos preliminares de arquitetura do sistema.

Conforme pode ser observado na Figura 18, e seguindo a técnica de transformação de modelos do tipo M2M da MDE, a AST consiste de um motor de transformação que importa um modelo funcional, executa um conjunto de regras de transformação previamente definido e gera como resultado um modelo de arquitetura preliminar do SCE, modelo este, hierarquicamente estruturado conforme o modelo funcional importado. Tanto o modelo funcional importado, quanto o modelo de arquitetura gerado, devem estar em conformidade com seus respectivos metamodelos.

A estrutura da AST apresentada na Figura 18 segue o esquema de transformação de modelos sugerido por (LINA; NANTES, 2006) e apresentado na Figura 5 do Capítulo 2 deste documento.

As regras de transformação da AST executam mapeamentos com multiplicidade de *um para um* **(1:1)** ou de *um para muitos* **(1:n)**. Os mapeamentos da AST podem ser classificados como sendo do tipo *combinado*, ou seja, eles combinam características do mapeamento de *tipos* e do mapeamento de *instâncias*. No mapeamento *combinado*, um mapeamento de *tipos*, pode ser, de certa forma, configurado através do uso de marcas. Neste caso, as marcas indicam características presentes no modelo funcional que não poderiam ser definidas através dos tipos de elementos existentes no modelo de arquitetura. Até este ponto da pesquisa, as regras de transformação da AST são unidirecionais, ou seja, elas tratam do mapeamento partindo de um modelo funcional para um modelo de arquitetura.

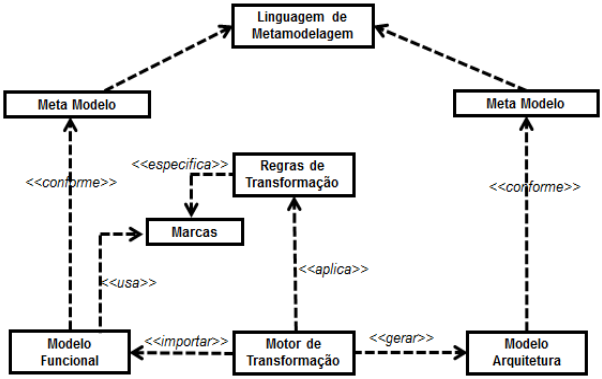

Figura 18 – Estrutura do Mecanismo de Transformação ASSsistida de Modelos.

As regras de transformação da AST oferecem suporte para a extração de aspectos estruturais, comportamentais (comportamento de determinados componentes de software) e de possíveis modos de operação do sistema do modelo funcional para gerar o modelo de arquitetura do SCE. A identificação destes diferentes aspectos se baseia exclusivamente na posição hierárquica de certos componentes dentro do modelo funcional e na composição interna destes componentes. A questão do mapeamento comportamental e dos modos de operação é melhor detalhada no próximo capítulo, o qual apresenta a utilização a AST para transformar modelos funcionais Simulink em modelos de arquitetura AADL.

A decisão pela utilização do recurso de marcas foi tomada durante a especificação das regras de transformação da AST. Ou seja, foi constatado que um diagrama de blocos no modelo funcional poderia gerar diferentes componentes de software ou de hardware no modelo de arquitetura do SCE durante o processo de transformação de modelos previsto. As variações de mapeamento identificadas estão diretamente relacionadas com a posição hierárquica do diagrama de blocos dentro do modelo funcional e com a composição interna do mesmo. Portanto, devido as variações de mapeamento identificadas durante a especificação das regras de transformação foi necessário definir algum tipo de interação do usuário com o processo de transformação de modelos previsto. A interação do usuário com o processo de transformação de modelos da AST ocorre, justamente, por meio da inserção das marcas nos blocos que possuem internamente um diagrama de blocos, diagrama

este, responsável pelo funcionamento de determinada funcionalidade do sistema. A necessidade da interação do usuário através da inserção das marcas motivou a denominar o mecanismo proposto como Transformação "ASssistida" de modelos (AST). Vale observar que o conjunto de marcas que pode ser utilizado para marcar um modelo funcional, e consequentemente, guiar a execução das regras de transformação da AST, deve ser previamente definido. Observe que o conjunto de marcas aparece representado na Figura 18 através da caixa marcas.

4.3 CORRELAÇÃO PROPOSTA ENTRE OS MODELOS FUNCIONAL E DE ARQUITETURA

Qualquer regra de transformação deve ser baseada em correlações existentes entre componentes dos modelos (e metamodelos) envolvidos no processo de transformação de modelos. Como pode ser observado na Figura 19, um modelo funcional pode possuir diferentes níveis de abstração, mas em última instância um modelo funcional gera o código-fonte de uma função do sistema. Este código normalmente é gerado a partir de um diagrama de blocos. Este diagrama de blocos, por sua vez, pode estar alocado hierarquicamente dentro de outro diagrama de blocos, e assim sucessivamente. É justamente esta estruturação hierárquica que possibilita a identificação de aspectos arquiteturais em um modelo funcional.

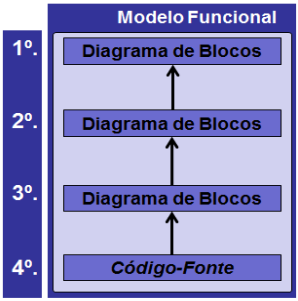

Figura 19 – Estrutura Hierárquica de um Diagrama de Blocos.

Visando explorar a possibilidade de estruturação dos modelos funcionais apresentada acima, o mecanismo de AST define que os modelos funcionais que serão importados pelo motor de transformação, devem, obrigatoriamente, descrever a hierarquia estrutural do sistema de controle por meio de subsistemas, suas interfaces e conexões. Sendo que cada subsistema deve estar especificado no modelo funcional como

um diagrama de blocos. Ferramentas de modelagem funcional bastante conhecidas como Simulink, Scade, LabVIEW e Scilab fazem uso do conceito de diagrama de blocos para estruturar seus modelos.

Para facilitar a compreensão da correlação entre os modelos funcional e de arquitetura sugerida por este trabalho de pesquisa, considere modelos funcionais gerados com o uso da ferramenta Simulink e modelos de arquitetura especificados utilizando a linguagem AADL. De acordo com a Figura 20, em um modelo funcional Simulink cada bloco *Subsystem* possui internamente um diagrama de blocos, que agrupa blocos de controle responsáveis pelo funcionamento de determinada função do sistema. Em AADL, o componente *Subprogram* é o componente que está associado ao código-fonte de uma função do sistema (através de uma propriedade específica que aponta para o nome do arquivo do código-fonte). Este componente, por sua vez é invocado ou chamado por um outro componente AADL chamado *Thread*. Em um modelo AADL um componente de software do tipo *Thread* deve estar alocado obrigatoriamente dentro de um componente AADL do tipo *Process*, e um componente *Process* deve estar alocado dentro de um componente *System*. Como pode ser observado na Figura 20 a estruturação hierárquica dos modelos funcional e de arquitetura possibilitam a identificação de correlações estruturais equivalentes entre eles.

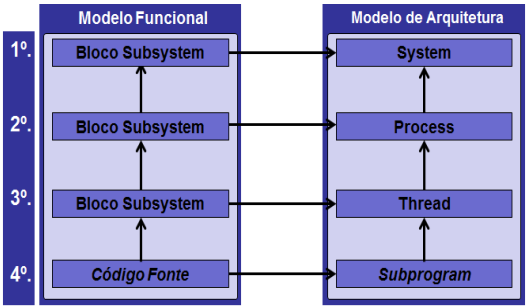

Figura 20 – Correlação ente os Modelos Funcional e de Arquitetura.

4.4 DIRETRIZES DE MODELAGEM BÁSICAS

No nível mais alto de abstração, o modelo de arquitetura gerado pela AST deve apresentar os dispositivos que interagem com o sistema, e nos níveis mais baixos de abstração, os componentes de software

do modelo de arquitetura do sistema. Porém, é importante destacar que a AST somente poderá extrair as informações esperadas de um modelo funcional se este estiver corretamente estruturado. Ou seja, o modelo funcional precisa expressar a hierarquia funcional do CPS. Uma hierarquia funcional deve expressar a hierarquia estrutural dos subsistemas que compõem o sistema de controle, suas interfaces e conexões. Este modelo funcional também pode ser usado durante o processo de desenvolvimento de um CPS para transmitir os requisitos do sistema para os responsáveis por projetar ou implementar os componentes do modelo funcional, caso estes ainda não estejam prontos. Os componentes de um modelo funcional (blocos que possuem internamente uma hierarquia de diagrama de bloco) correspondem a outros modelos funcionais.

Sendo assim, esta seção apresenta algumas diretrizes de modelagem para os modelos funcionais que serão utilizados como modelos de entrada para a AST. A não aplicação das diretrizes de modelagem por parte do usuário pode dificultar a identificação de aspectos estruturais no modelo funcional, e como consequência, o modelo de arquitetura gerado pela AST pode não refletir de forma satisfatória a arquitetura SCE. Ou seja, o modelo de arquitetura vai ser gerado pela AST independentemente da aplicação das diretrizes de modelagem no modelo funcional importado, entretanto, neste caso, o modelo de arquitetura gerado pela AST pode requerer uma quantidade maior ajustes adicionais.

As diretrizes de modelagem recomendadas para os modelos funcionais que serão importados pela AST são as seguintes:

- **Diretriz de Modelagem 1:** Os blocos de controle responsáveis por funções ou sub-funções do sistema devem estar obrigatoriamente agrupados em blocos que representem subsistemas.
- **Diretriz de Modelagem 2:** Os tipos das portas de um bloco que representa um subsistema, e o tipo dos dados que podem ser recebidos ou transmitidos por estas portas devem estar corretamente definidos.
- **Diretriz de Modelagem 3:** O bloco ou o conjunto de blocos que venham a representar um sensor ou atuador no sistema de controle, devem, obrigatoriamente, estar sub-alocados em um bloco do tipo subsistema.
- **Diretriz de Modelagem 4:** Não utilizar acentos e espaços entre palavras para nomear blocos do tipo subsistema e as portas de

entrada e saída destes blocos.

As diretrizes de modelagem básicas definidas nesta seção tem como principal objetivo facilitar a identificação de aspectos estruturais no modelo funcional de um CPS, principalmente a diretriz de modelagem 1. Como pode ser observado, as diretrizes apresentadas são bastante simples e não restringem a criação de um modelo funcional. Além disso, alguns manuais fornecidos pelos fabricantes das ferramentas de modelagem, como por exemplo o Simulink, recomendam a utilização de blocos do tipo subsistema (*Subsystem*) para a organizar hierarquicamente os modelos.

Dependendo do par *ferramenta de modelagem funcional/linguagem de descrição de arquitetura* utilizado durante o processo de desenvolvimento de um CPS, pode surgir a necessidade da definição de novas diretrizes de modelagem. A necessidade de definição de diretrizes de modelagem adicionais está diretamente relacionada a quantidade de recursos oferecidos ou pela ferramenta de modelagem funcional ou pela linguagem de descrição de arquitetura adotados.

4.5 ESTRUTURA HIERÁRQUICA GENÉRICA DE UM MODELO DE ARQUITETURA GERADO PELA AST

A Figura 21 apresenta a estrutura hierárquica genérica de um modelo de arquitetura gerado pela AST. O modelo de arquitetura gerado pela AST é uma árvore de componentes hierarquicamente estruturados. Espera-se que o modelo de arquitetura gerado pela AST possa ser instanciado através de uma hierarquia de componentes, partindo de um elemento raiz (*system*) para as folhas (*subprograms*).

É importante compreender que o modelo de arquitetura gerado pela AST é um modelo preliminar. Ou seja, o projetista precisa, posteriormente, especificar uma plataforma de execução que suporte a execução dos componentes da aplicação, e também inserir informações adicionais nos componentes do modelo de arquitetura gerado, para que ele possa então, ser submetido a análises e verificações, e assim ser validado. É por este motivo, que os componentes *processor*, *memory* e *bus* aparecem pontilhados na figura 21.

Uma característica da AST que tende a agilizar consideravelmente o trabalho do projetista e que contribui com a consistência e integração entre os modelos funcional e de arquitetura de um CPS, é o fato dela gerar um sistema raiz que pode ser diretamente instanciado, composto por componentes *subgram*, chamadas de subprogramas dentro

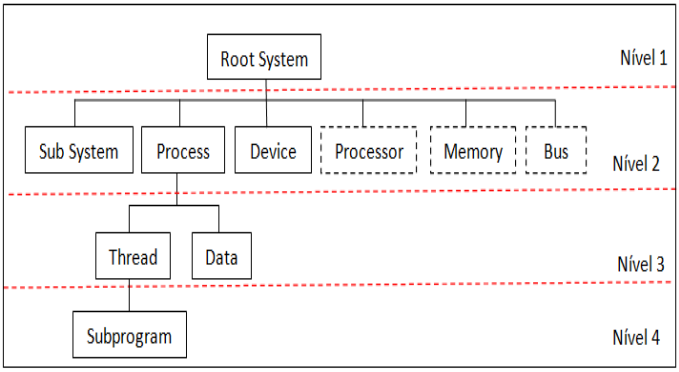

Figura 21 – Estrutura Hierárquica Genérica de Modelo de Arquitetura gerado pela AST

das *threads*, e que possui os tipos de dados das portas do tipo *data port* e *event data port* devidamente especificados e consistentes como modelo funcional.

Conforme já comentado, a interação do usuário com o processo de transformação de modelos da AST ocorre, basicamente, por meio da atividade de inserção das marcas nos blocos que possuem internamente diagramas de bloco e que representam dispositivos externos, como sensores ou atuadores, no modelo funcional, ou ainda, pela inserção ou edição das marcas em outros blocos que possuam internamente diagramas de bloco. As marcas representam um conceito similar a um esteriótipo UML. Uma vez que a interação do projetista esteja concluída, o motor de transformação da AST é capaz de gerar o modelo de arquitetura preliminar.

4.6 VALIDAÇÃO DE UM MODELO DE ARQUITETURA GERADO PELA AST

A validação de um modelo de arquitetura de sistema pode ser feita de diversas formas. Validar a sintaxe é evidentemente necessário, mas além disso é interessante validar outros aspectos do sistema, como por exemplo, segurança, latência, escalonabilidade, etc. Entretanto, estas análises requerem que o modelo de arquitetura contenha uma plataforma de execução definida (processadores, memória, etc.).

Como já mencionado, o objetivo da AST é gerar um modelo

de arquitetura do sistema a partir de informações extraídas de um modelo funcional. Entretanto, posteriormente, o projetista precisa definir a plataforma de execução do sistema, ou seja, ele precisa definir o hardware que suporta a execução dos componentes da aplicação gerados pela AST, e o hardware que suporta as conexões lógicas entre os componentes da aplicação.

Antes de submeter a instância de um modelo de arquitetura gerado pela AST a análises e verificações, também é necesário que o projetista especifique algumas propriedades de execução dos componentes de software, tais como: *dispatch_protocol*, *period*, *deadline* e *compute_exectuion_time*. Além de também especificar as políticas de escalonamento e a capacidade de processamento do processador. Depois disso, e uma vez tendo estabelecido as ligações entre os componentes de software com o(s) processador(es) é possível avaliar a taxa de utilização do(s) processador(es) e fazer uma análise de escalonamento para determinar se as ligações existentes são aceitáveis. Também é possível atribuir orçamentos de recursos em termos de ciclos de CPU e memória para os subsistemas da aplicação e capacidades de recursos para a plataforma de execução. Diante desses dados, o projetista pode analisar diversas ligações entre os componentes da aplicação e a plataforma de execução, e garantir que os orçamentos não excedem a capacidades específicas da plataforma definida.

Para que o modelo de arquitetura gerado pela AST possa ser submetido a análises de fluxo, como à análise de tempo de resposta, o projetista precisa adicionar especificações de fluxo nos componentes do sistema, e especificar o fluxo *end-to-end* no sistema raiz do modelo de arquitetura.

Para submeter o modelo de arquitetura gerado pela AST à verificação formal de propriedades comportamentais, e assegurar a corretude do comportamento do modelo, é necessário que antes disso o projetista especifique o comportamento dos *devices* e das *threads* utilizando para isso os recursos disponíveis pela linguagem de descrição de arquitetura que está sendo utilizada. Dentre os pontos passíveis de verificação, destacam-se, por exemplo, as propriedades comportamentais das *threads*, tais como vivacidade, ausência de bloqueio e justiça. Neste caso, o mapeamento comportamental da AST agiliza a especificação destes comportamentos, uma vez que algumas *threads* do modelo de arquitetura gerado pela AST podem possuir seu comportamento previamente especificado.

Enfim, a AST gera um modelo de arquitetura preliminar, apto a receber informações como a especificação de uma plataforma de

execução que suporte a execução dos componentes da aplicação, e também a inserção informações adicionais nos componentes do modelo de arquitetura para que então, o referido modelo possa ser submetido a diversos tipos de análises e verificações não disponíveis para modelos funcionais.

4.7 CONSIDERAÇÕES

Considerando que os engenheiros estão habituados a utilizar ferramentas de modelagem funcional para demonstrar os aspectos funcionais de seus sistemas através de uma hierarquia de diagramas de blocos, a AST oferece suporte para que as ferramentas de modelagem funcional possam atuar como um *front-end* para gerar modelos de arquitetura preliminares.

A AST combina de forma coerente alguns recursos de mapeamento e princípios de correlação entre os modelos sugeridos pelas abordagens de mapeamento entre modelos funcionais e de arquitetura apresentadas por (RAGHAV et al., 2009b) e (DELANGE et al., 2010). Ou seja, assim como acontece em (RAGHAV et al., 2009b), a AST explora a capacidade que as ferramentas de modelagem possuem de organizar seus projetos utilizando diagramas de blocos hierárquicos, e as regras de transformação da AST também são guiadas por marcas inseridas em determinados componentes do modelo funcional.

Além disso, a AST mapeia os tipos de dados que trafegam pelas portas dos blocos do modelo funcional em componentes de software do tipo *data* no modelo de arquitetura, e também mapeiam para cada função do sistema como um componente de software do tipo *thread* que chama internamente outro componente do tipo *subprogram*, assim como é sugerido pela abordagem apresentada por (DELANGE et al., 2010).

5 APLICAÇÃO DA AST NA TRANSFORMAÇÃO DE MODELOS FUNCIONAIS SIMULINK EM MODELOS DE ARQUITETURA AADL

Para avaliar o mecanismo de transformação assistido de modelos proposto por este trabalho de pesquisa, foi definido um par *ferramenta de modelagem funcional/linguagem de descrição de arquitetura* para gerar, respectivamente, os modelos de entrada e de saída do motor de transformação de modelos proposto.

O Simulink foi escolhido como a ferramenta de modelagem funcional por ser uma ferramenta comumente utilizada para projetar algoritmos e técnicas de controle de sistemas embarcados de diversos domínios de aplicação, tais como, automotivo, metroviário, industrial e aeroespacial. Como linguagem de descrição de arquitetura foi escolhida a linguagem AADL, ela foi escolhida porque possibilita especificar adequadamente modelos que descrevem individualmente a arquitetura de software, a plataforma de execução, composta por processadores, barramentos e memória, e a integração entre estes dois modelos para compor o modelo de arquitetura completo do sistema embarcado, o que não está disponível no mesmo grau de detalhe para os modelos Simulink. Além disso, a linguagem AADL possui ferramentas de suporte adequadas para realizar diferentes tipos de análises e verificações no modelo de arquitetura do sistema, as quais também não estão disponíveis para modelos Simulink [(CORREA et al., 2010), (JENSEN; CHANG; LEE, 2011)].

A figura 22 apresenta uma adaptação do processo de transformação da AST apresentado na figura 18, considerando, neste caso, a transformação assistida de modelos entre um modelo funcional Simulink e um modelo de arquitetura AADL.

Considerando a transformação assistida de modelos entre os modelos Simulink e AADL, primeiro foi preciso especificar e compreender os metamodelos Simulink e AADL. Na sequência, foi preciso especificar o conjunto de marcas que pode ser utilizado para marcar o modelo funcional Simulink e especificar o conjunto de regras de transformação propriamente dito. Para então, ser possível implementar o motor de transformação capaz de importar um modelo funcional Simulink, aplicar de forma automatizada as regras de transformação definidas anteriormente e gerar o modelo AADL preliminar.

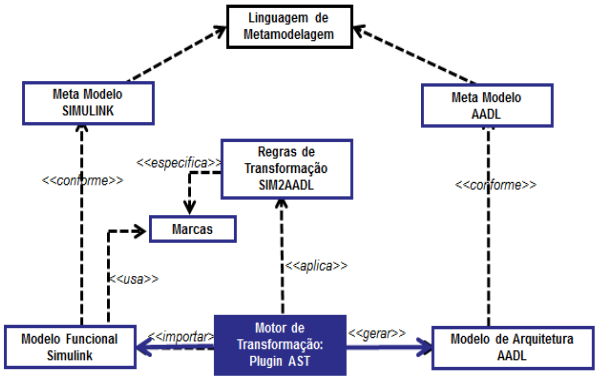

Figura 22 – Estrutura do Processo da Transformação ASSsistida de modelos Simulink/AADL.

5.1 METAMODELOS

Um metamodelo normalmente é representado como um diagrama de classes UML e define a estrutura, a semântica e as retrições de uma família de modelos. Os metamodelos são elementos muito importantes no processo de transformação de modelos da AST, uma vez que eles desempenham o papel de orientar o desenvolvimento dos modelos fonte e alvo, e também, porque guiaram a especificação das regras de transformação, do conjunto de marcas e das diretrizes de modelagem adicionais.

5.1.1 Metamodelo Simulink

Como não há uma versão oficial do metamodelo Simulink disponível, e como os metamodelos apresentados em (NEEMA, 2001), (LEGROS et al., 2007) e (BOSTRÖM et al., 2010) se mostraram insuficientes para estruturar as regras de mapeamento da AST, foi necessário especificar um novo metamodelo Simulink para atender aos objetivos da AST. O metamodelo Simulink proposto fornece uma descrição compacta dos aspectos estruturais de modelos Simulink.

A Figura 23 apresenta o metamodelo Simulink proposto no escopo deste trabalho.

Os componentes do metamodelo Simulink e suas relações são descritos brevemente a seguir:

Model: É o componente raiz em um arquivo mdl, e por

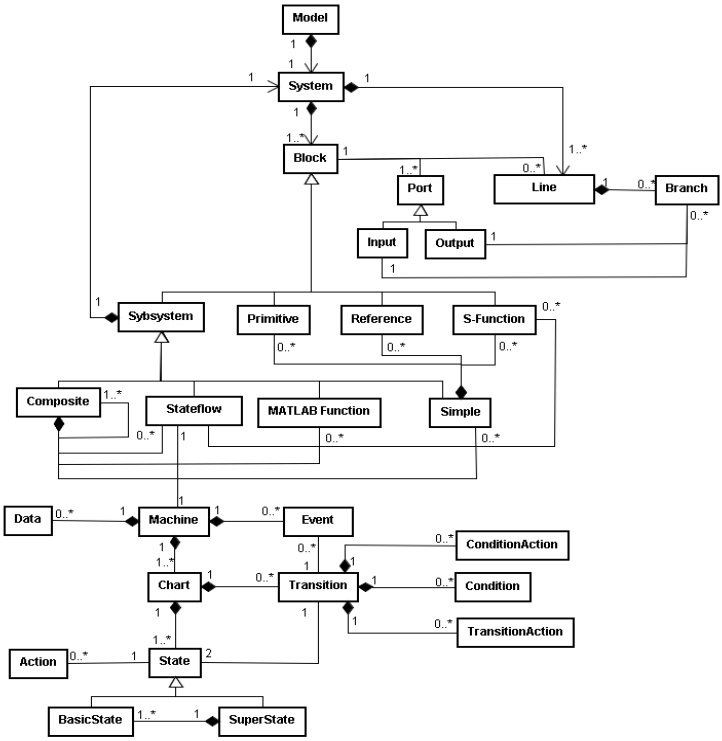

Figura 23 – Metamodelo Simulink

consequência o elemento raiz do metamodelo Simulink. Cada projeto Simulink contém uma única instância do componente *Model*. Ele captura opções gerais de um projeto Simulink, as quais incluem: preferências visuais, preferências de impressão, informações sobre o projeto, opções de simulação/*solver*, opções de geração de código, etc. O componente *Model* é composto por um único componente *System*.

System: Este componente encapsula um modelo Simulink, e serve como um *container* para um diagrama de blocos ou para um modelo dinâmico. O componente *System* é composto por um ou mais componentes *Block* e por componentes *Line*.

Block: É o principal elemento do componente *System*. Ele representa um componente funcional do modelo. Um bloco pode ser especializado como sendo do tipo:

1. Subsystem: são blocos que agrupam blocos de controle de

uma mesma funcionalidade. Um bloco do tipo *Subsystem* é composto por um outro diagrama de blocos. Esta composição é indireta através de um componente *System*, e isto, é o que permite a representação hierárquica de um sistema complexo. Um bloco do tipo *Subsystem* pode ser do tipo:

1.1 Composite: são blocos compostos por outros blocos *Subsystem* do tipo *Composite*, *Stateflow*, *MATLAB Function*, e/ou *Simple*.

1.2 Stateflow: este bloco representa o comportamento dinâmico de um sistema ou subsistema através de um diagrama Stateflow.

1.3 MATLAB Function: são blocos capazes de armazenar funcionalidades do sistema escritas em linguagens de programação, tais como MATLAB, C, C++ ou Fortran.

1.4 Simple: são blocos que podem possuir internamente blocos do tipo primitive, S-Function e Reference.

2. Primitive: são blocos que estão disponíveis nas bibliotecas do Simulink.

3. Reference: são blocos criados pelo próprio usuário e que podem ser reutilizados em diferentes modelos Simulink.

4. S-Function: são blocos que permitem o acesso a S-Functions escritas em MATLAB, C, C++ ou Fortran a partir de um diagrama de blocos.

Line: é um componente que conecta os blocos de um modelo através de suas portas de entrada e de saída.

Port: define a interface dos blocos. As portas de entrada e saída de um bloco *Subsystem* são especificadas respectivamente por blocos primitivos *Inport* e *Outport*.

Machine: representa o componente raiz na hierarquia Stateflow. Pode haver no máximo um elemento *Machine* por projeto Simulink. Este componente pode conter componentes *Event* e *Data* de diferentes componentes *Chart*.

Chart: representa o gráfico da máquina de estados finito. Pode haver vários componentes *Chart* associados a um componente *Machine*.

State: são os componentes básicos de um Stateflow, e representam os modos de operação de um sistema dinâmico. Estados podem estar ativos ou inativos, estado ativo significa que o Stateflow está nesse modo de operação. Existem dois tipos de composição semântica para os estados. A composição *sequencial*, onde apenas um estado pode estar ativo por vez, e a composição *paralela*, onde vários estados podem estar ativos em um dado momento, representando assim o paralelismo. Cada estado do Stateflow pode estar associado a *Actions*. No arquivo mdl

o elemento *Action* é uma propriedade identificada como *labelString* no estado. O componente *State* pode ser especializado em dois tipos, sendo eles:

1- **BasicState:** este componente representa um estado simples de uma máquina de estados;

2- **SuperState:** este componente é o responsável por oferecer suporte à composição hierarquica de máquinas de estado, ou seja, são estados que contêm outros diagramas *Stateflow*.

Event: este componente representa um mecanismo de disparo. Ele pode estar associado ao componente *Machine*, condicionando a ativação do mesmo a determinado evento. Um componente *Event* também pode estar associado a uma transição, indicando quando a mesma deve disparar. Caso exista mais de um evento que possa disparar um *Machine* ou uma *Transition*, é possível utilizar o operador lógico "ou".

Data: este componente representa informações. Este componente e o componente *Event* são utilizados pelos blocos Simulink e Stateflow para o intercâmbio de informações.

Transition: este componente representa a passagem de um estado de origem para um estado destino. Transições podem ser disparadas por eventos, e possuir condições de guarda, ações de condições, e ações de transição.

Condition: este componente está diretamente ligado ao componente *Transition*, uma vez que ele impõe uma condição que deve ser satisfeita para que a transição seja disparada. No arquivo mdl o elemento *Condition* é uma propriedade identificada como *labelString* na transição. Caso exista mais de uma condição para habilitar o disparo da transição é possível utilizar os operadores lógicos "e , ou" para fazer a combinação. Ela aparece entre colchetes [ ] no rótulo de uma transição no diagrama Stateflow.

ConditionAction: representa uma ação que deve ser executada assim que a condição é avaliada como verdadeira, antes do destino da transição ser alcançado. Se nenhuma condição for especificada na transição, uma condição implícita é avaliada como verdadeira e a condição da ação é executada. Ela aparece entre colchetes { } após a condição no rótulo de uma transição no diagrama Stateflow.

TransitionAction: representa uma ação que deve ser executada após o destino da transição ter sido alcançado, desde que a condição, se especificada, seja verdadeira. A ação de uma transição aparece depois de uma condição de ação antecedida com uma barra invertida (/) no rótulo de uma transição no diagrama Stateflow.

5.1.2 Metamodelo AADL

Durante a estruturação da AST o metamodelo AADL foi utilizado como referência para definir o conjunto de marcas, o qual é utilizado pelo projetista para marcar um modelo funcional Simulink, e também para estruturar as regras de transformação e possiblitar a criação do modelo da arquitetura de software em AADL, o qual é considerado o modelo alvo do processo de transformação de modelos da AST. Como a versão oficial do metamodelo AADL usada pelo OSATE e disponível em (SAE, 2006), possui pelo menos 100 entidades e está subdividido em diferentes arquivos ECORE, foi gerada uma versão simplificada do metamodelo oficial da AADL para facilitar a leitura e o entendimento do mesmo. A Figura 24 apresenta a versão simplificada do metamodelo oficial da AADL. A parte referente ao anexo comportamental da versão simplificada do metamodelo AADL, apresentado na figura 24, foi criada utilizando como referência o metamodelo apresentado em (LASNIER et al., 2011) e na interpretação do AADL Behavior Annex (SAE, 2011a).

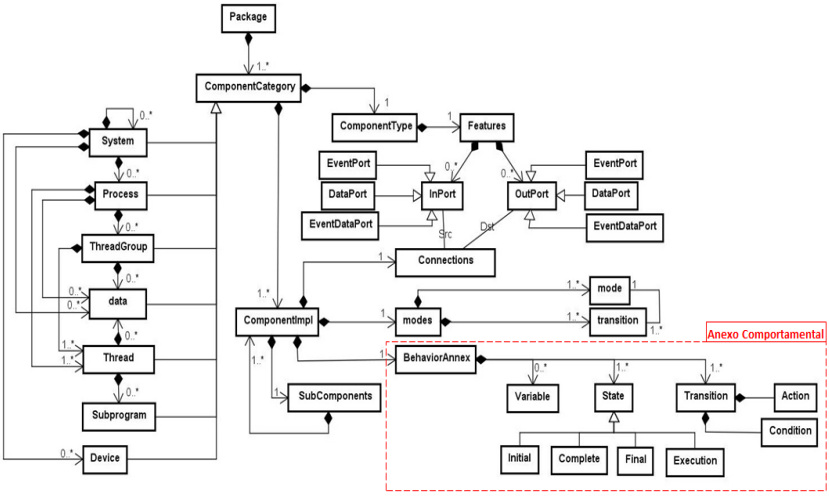

Figura 24 – Versão Simplificada do Metamodelo da AADL

Os componentes do metamodelo AADL e suas relações são descritos brevemente a seguir:

Package: todo os componentes que compõem um modelo AAADL ficam alocados dentro de um *Package*. *Packages* permitem

a organização dos elementos.

ComponentCategory: este componente pode ser especializado nos componentes: *System, Process, GroupThread, Thread, Data, Subprogram, Device*. Um modelo AADL possui vários destes componentes para descrever um sistema.

ComponentType e ComponentImpl: os componentes de um modelo AADL são definidos através de declarações *ComponentType* e *ComponentImplementation*. Uma declaração *ComponentType* define a categoria e as interfaces (*features*) do componente. Uma declaração *ComponentImplementation* define a estrutura interna de um componente em termos de subcomponentes, conexões com subcomponentes, modes e comportamento.

Connections: representa os componentes responsáveis pela comunicação entre diferentes *ComponentType* de um modelo AADL. Um determinado *ComponentImpl* pode receber ou enviar dados, eventos e dados/eventos através de portas de entrada e saída.

Modes: este componente permite especificar os modos de funcionamento ou estados operacionais de qualquer *ComponentType* de um modelo AADL. Os modos de operação são representados por estados dentro de uma abstração de máquina de estados, chamada de máquina de estados modal do componente. A subseção *modes* também contém eventos que possuem ou não condições, as quais causam as transições entre os modos de operação do componente.

System: este componente representa um sistema. Ele pode ser composto por um ou vários componentes *Process*, um ou vários componentes *data*, um o vários componentes *device*, e até mesmo por outros componentes *System*. Todo modelo AADL deve possuir um elemento *System* raíz.

Process: este componente representa um espaço de endereço protegido. Ele pode ser composto por nenhum, um ou vários componentes *ThreadGroup*; um ou vários componentes *Thread*; e um ou vários componentes *data*.

ThreadGroup: este componente representa uma unidade de composição para organização de *threads*. Desta forma, ele pode ser composto por um ou vários componentes *Thread*, por um ou vários componentes *data*, e até mesmo por outros componentes *GroupThread*.

Thread: este componente representa uma unidade escalonável de execução simultânea, ou seja, representa basicamente uma função ou tarefa a ser executada pelo sistema. Este componente pode ser composto por um ou vários componentes *data* e chamar um ou vários componentes *subprogram* .

Data: este componente representa os tipos de dados, e os dados estáticos do código fonte da aplicação. Ele pode ser composto apenas de outros componentes *data*.

Subprogram: este componente representa um código executável a ser chamado sequencialmente dentro de uma *thread*. Ele não suporta nenhum subcomponente.

BehaviorAnnex: é o componente raiz da descrição do comportamento de um *ComponentType*. O Anexo Comportamental da linguagem AADL permite descrever o comportamento de qualquer tipo de componente através de um sistema de transição de estados com condições de guarda e ações. O autômato estendido que representa o comportamento de um componente é basicamente composto por *variables, states,* e *transitions*. Se o comportamento estiver anexado diretamente ao tipo de componente, todos os componentes daquele tipo utilizados no modelo possuirão o mesmo comportamento. Se o comportamento estiver anexado especificamente a um componente (em sua *implementation*), o comportamento descrito será seguido apenas por aquele componente.

Variable: este componente representa variáveis locais. Estas variáveis podem ser utilizadas para manter o controle dos resultados intermediários dentro do escopo do componente *BehaviorAnnex*, elas são espaços reservados para dados do aplicativo que podem requerer armazenamento temporário.

State: representam os estados do autômato utilizado para representar o comportamento do componente (*System, Process, Thread, Subprogram, etc.*) no modelo AADL. O componente *State* pode ser especializado como *initial ,complete, final, execution*, ou suas combinações. Um estado sem tipo definido será considerado automaticamente um *estado de execução*. O comportamento de um autômato deve começar a partir de um estado inicial e termina em um estado final.

Transition: este componente define as transições de um estado fonte para um estado destino. Um componente *Transition* pode estar associado a uma *Condition* e opcionalmente a uma *Action*.

Condition: este componente determina se uma transição é disparada.

Action: este componente especifca a ação ou ações a serem executadas quando a condição for avaliada como verdadeira.

5.2 CORRELAÇÕES ENTRE OS METAMODELOS SIMULINK E AADL

De acordo com (FEILER; GLUCH, 2012) os modelos Simulink podem ser usados para complementar os componentes AADL, fornecendo suas funcionalidades. As funcionalidades em si não podem ser expressas usando a sintaxe AADL, elas são representadas como componentes do tipo *"black-box"* que são usados por outros componentes AADL. Na verdade, este componente do tipo *"black-box"* é normalmente um componente de software AADL do tipo *subprogram*, que normalmente está associado a (ou é invocado por) uma *thread* AADL. O que pode ser representado utilizando a sintaxe da AADL é o comportamento de uma determinada funcionalidade, utilizando para isso o anexo comportamental da AADL.

Em Simulink, tal especificação funcional está presente em qualquer bloco *Subsystem* do tipo *Composite*, *Simple*, *Stateflow* ou *MatlabFunction*. Portanto, é possível considerar que os blocos *Subsystem*, com suas respectivas portas e conexões, podem ser mapeados para uma estrutura hierárquica AADL *System-Process-Thread-Subprogram*. Isso constitui o que é considerado pela AST como uma *"correlação âncora"*. A partir desta correlação, as demais correlações foram especificadas utilizando uma navegação de cima para baixo nos metamodelos. A Figura 25 apresenta graficamente a correlação descrita.

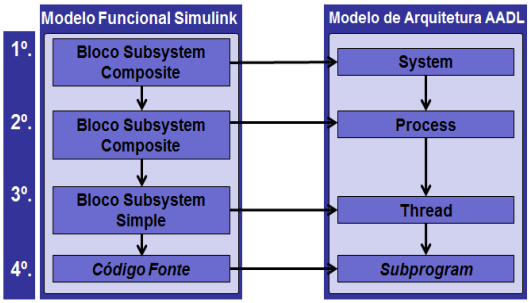

Figura 25 – Correlação ente os Modelos Simulink e AADL.

Em AADL, os *Devices* representam elementos externos que interagem com o sistema, como sensores, atuadores, HMI ou toda a planta do sistema sob controle. Este tipo de elemento normalmente existe no modelo Simulink, mas deve ser devidamente identificado pelo projetista através da inserção de marcas.

O bloco Simulink *Stateflow* pode ser mapeado para AADL como sendo os modos de operação do sistema, ou então, como a especificação do comportamento interno de uma *Thread*. Ele representa os modos de operação do sistema quando ele está localizado em um nível hierárquico mais elevado dentro do modelo Simulink. Como esperado, ele representa o comportamento interno de uma *Thread* quando está localizado em níveis hierárquicos inferiores do modelo Simulink.

As correlações observadas só se aplicam quando se considera um projeto de modelos Simulink hierárquico. Obviamente, se um modelo Simulink plano é criado, não é possível estabelecer tais correlações. Portanto, para que os projetistas utilizem a AST, os engenheiros de controle devem obrigatoriamente, trabalhar com modelos hierárquicos. Em outras palavras, isto significa que as restrições de modelagem apresentadas no capítulo anterior devem ser satisfeitas pelo modelo funcional Simulink.

5.3 DIRETRIZES DE MODELAGEM ADICIONAIS

Devido aos recursos específicos oferecidos pela ferramenta de modelagem funcional Simulink e pela linguagem de descrição de arquitetura AADL, foram definidas algumas diretrizes de modelagem adicionais, além das diretrizes de modelagem básicas apresentadas no capítulo anterior. Tais diretrizes facilitam a identificação e o posterior mapeamento dos modos de operação do sistema e do comportamento de outros componentes do sistema do modelo funcional Simulink para o modelo de arquitetura AADL.

- **Diretrizes de Modelagem para Especificação de Diagramas Stateflow que Expressam os Modos de Operação de um Sistema de Controle**

  Durante a especificação das regras de transformação da AST foi preciso definir algumas diretrizes de modelagem adicionais para tornar possível o mapeamento dos modos de operação, uma vez que a linguagem utilizada para expressar diagramas Stateflow possui um conjunto de estruturas e recursos muito maior do que os disponíveis para expressar os modos de operação de componentes em modelos AADL. Sendo assim, as diretrizes de modelagem a seguir também precisam ser satisfeitas pelos modelos funcionais Simulink para que o mapeamento operacional da AST possa extrair as informações desejadas.

- **Diretriz de Modelagem 5:** O bloco *Subsystem* do tipo *Stateflow* que possui o papel de especificar os modos de operação do sistema deve estar posicionado, obrigatoriamente, no primeiro ou segundo nível hierárquico do modelo funcional Simulink.
- **Diretriz de Modelagem 6:** O diagrama *Stateflow* do bloco *Subsystem* do tipo *Stateflow* que especifica os modos de operação do sistema, deve ser composto por estados básicos e exclusivos, por ações de estado, e por transições de estado sem condições de guarda, e exclusivamente desencadeadas por eventos. As transições de estado não podem possuir ações de condição e ações de transição. Qualquer outra estrutura disponível para diagramas Stateflow deve ser evitada. As ações de estado devem especificar os blocos *Subsystem* a serem ativados no momento que determinado estado (modo de operação) estiver ativo.
- **Diretriz de Modelagem 7:** As portas de entrada e saída de um bloco *Subsystem* do tipo *Stateflow* que recebeu a marca *mode* devem receber e enviar exclusivamente eventos.

- **Diretrizes para Especificação de Diagramas Stateflow que Expressam o Comportamento Interno de uma Funcionalidade do Sistema de Controle**

  A exemplo do que aconteceu com o mapeamento dos modos de operação, durante a especificação das regras de transformação da AST também foi preciso definir algumas restrições para tornar mais seguro o mapeamento comportamental. Neste caso, para que não fosse necessário reescrever alguns elementos em um diagrama Stateflow antes de realizar o mapeamento comportamental, optamos por inicialmente fazer o mapeamento de apenas um subconjunto de estruturas. Sendo assim, mais duas diretrizes de modelagem precisam ser satisfeitas pelos modelos funcionais Simulink para que o mapeamento comportamental da AST possa extrair as informações desejadas.

  - **Diretriz de Modelagem 8:** O diagrama Stateflow do bloco *Subsystem* do tipo *Stateflow* que possui o papel de especificar o comportamento interno de uma *thread*, deve estar posicionado no último nível hierárquico do modelo funcional Simulink.

– **Diretriz de Modelagem 9:** O diagrama *Stateflow* do bloco *Subsystem* do tipo *Stateflow* que especifica o comportamento de uma thread deve ser composto apenas por estados básicos e exclusivos, por ações de estado, e por transições de estado, com ou sem condições de guarda, as quais podem ser combinadas usando cojunções(&&) ou disjunções(|), e desencadeadas ou não por diferentes eventos separados pelo operador lógico | (ou). As transições podem possuir ações de transição, mas não devem possuir ações de condição. Os estados podem possuir fluxogramas. Qualquer outra estrutura disponível para diagramas Stateflow deve ser evitada.

5.4 CONJUNTO DE MARCAS

As *marcas* representam conceitos AADL e são utilizadas para marcar os blocos *Subsystem* no modelo funcional, elas indicam as decisões de projeto tomadas pelo projetista. A Figura 26 apresenta o conjunto de marcas que pode ser utilizado pelo projetista para marcar os blocos *Subsystem* do modelo funcional Simulink. A primeira coluna indica os tipos de blocos *Subystem* que podem ser marcados no modelo funcional Simulink, a segunda coluna apresenta as marcas que estes blocos podem receber, e a terceira coluna apresenta a posição hierárquica que o bloco *Subsystem* precisa ocupar dentro modelo funcional Simulink para receber a respectiva marca.

Para facilitar a compreensão do papel que as marcas desempenham no processo de transformação de modelos da AST, considere as situações descritas a seguir:

- Quando a marca *system* é inserida em um bloco *Subsystem* do tipo *Composite* que está no primeiro nível hierárquico do modelo funcional Simulink, o motor de transformação da AST gera no modelo AADL resultante um componente *system* de mesmo nome. Porém, é importante o projetista ter consciência de que os blocos *Subsystem* pertencentes a ele, devem, obrigatoriamente, receber as marcas *system*, *process* ou *process/thread* para que o modelo AADL gerado pelo motor de transformação possa ser considerado sintaticamente válido.

- Quando a marca *process/thread* é inserida em um bloco *Subsystem* do tipo *Composite* que está posicionado em qualquer nível

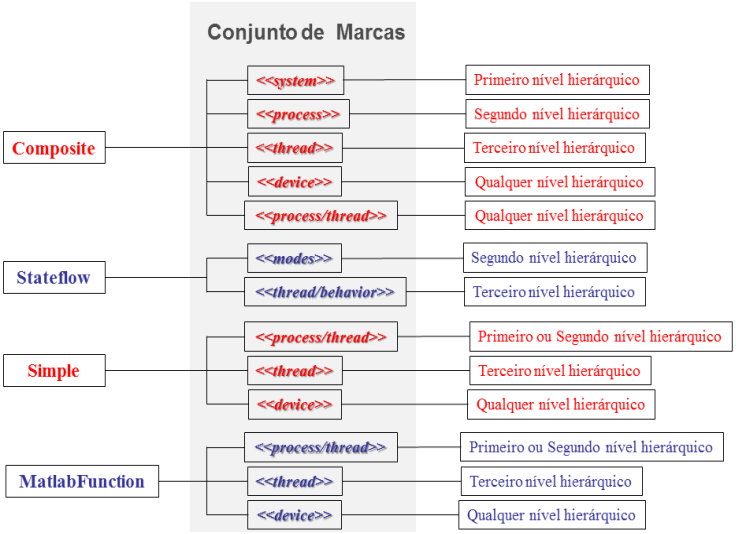

Figura 26 – Conjunto de Marcas - Marcação Manual.

hierárquico do modelo funcional Simulink, o projetista deve ter consciência de que os blocos *Subsystem* pertencentes a ele não devem receber qualquer tipo de marca, uma vez que estas serão ignoradas pelo motor de transformação da AST. Ou seja, quando um bloco *Subsystem* do tipo *Composite* recebe a marca *process/thread*, o motor de transformação da AST gera no modelo AADL resultante um componente de software do tipo *process*, que possui como subcomponente um único componente *thread*, que por sua vez possui a chamada à um componente *subprogram*.

- Quando a marca *process/thread* é inserida em um bloco *Subsystem* do tipo *Simple* que está posicionado no primeiro ou segundo nível hierárquico do modelo funcional Simulink, o motor de transformação da AST baseado no mapeamento estrutural gera no modelo AADL resultante um componente de software do tipo *process*, que possui um subcomponente *thread*, que por sua vez possui a chamada à um componente *subprogram*.

- Quando a marca *thread/behavior* é inserida em um bloco *Subsystem* do tipo *Stateflow* que está posicionado no terceiro nível hierárquico do modelo funcional Simulink, o motor de transformação da AST baseado no mapeamento comportamental gera no modelo

AADL resultante um componente de software *process*, que possui um subcomponente *thread*, que por sua vez possui seu comportamento especificado na seção *annex behavior_specification*.

As portas de entrada e saída de qualquer componente AADL gerado pelo motor de transformação da AST devem suportar o envio ou o recebimento do mesmo tipo de dado e devem possuir o mesmo nome das portas do bloco *Subsystem* do modelo funcional Simulink que o originou. Isto é o que garante a consistência entre as interfaces dos blocos *Subsystem* do modelo funcional Simulink e dos componentes de software do modelo AADL gerado pelo motor de transformação da AST.

Para adicionar as marcas nos blocos *Subsystem* do modelo funcional Simulink o projetista pode utilizar o recurso de inserção de anotações da própria ferramenta Simulink. Para isto, basta clicar com o botão direito do mouse sobre o bloco *Subsystem* que deseja marcar, selecionar a opção *Properties* e na guia *Block Annotation* digitar a marca desejada na caixa de diálogo *Enter text and tokens for annotation*, conforme indicado na Figura 27, e em seguida confirmar Ok. As marcas a serem utilizadas pelo projetista estão restritas ao conjunto pré-definido, apresentado na Figura 26, além disso, devem ser respeitadas as especificações de tipo de bloco *Subsystem* e a posição hierárquica destes blocos dentro do modelo funcional Simulink, como especificado previamente na mesma tabela.

Caso preferir, o projetista também pode utilizar o *Assistente de Marcação* que foi implementado no escopo deste trabalho. O *Assistende de Marcação* percorre o arquivo mdl e marca automaticamente os blocos *Subsystem* do modelo funcional Simulink de acordo com o tipo e a posição hierárquica dos mesmos, respeitando, neste caso, as especificações da Figura 28. A única marca não inserida pelo assistente de marcação é a marca *Device*. Portanto, o projetista precisa inserir a marca *Device* em qualquer bloco *Subsystem* que represente um dispositivo externo que interage com o sistema, como um sensor, atuador, HMI ou toda a planta do sistema sob controle. Este tipo de elemento normalmente existe em um modelo Simulink, mas deve ser devidamente identificado pelo projetista.

Quando um modelo funcional Simulink marcado de forma manual é importado pelo motor de transformação (*plugin AS2T*) as marcas são preservadas. Quando um modelo funcional Simulink que não foi previamente marcado é importado pelo *plugin AS2T*, o assistente de marcação do *plugin* insere automaticamente as marcas nos blocos *Subsystem* considerando o tipo e a posição hierárquica do mesmo. As

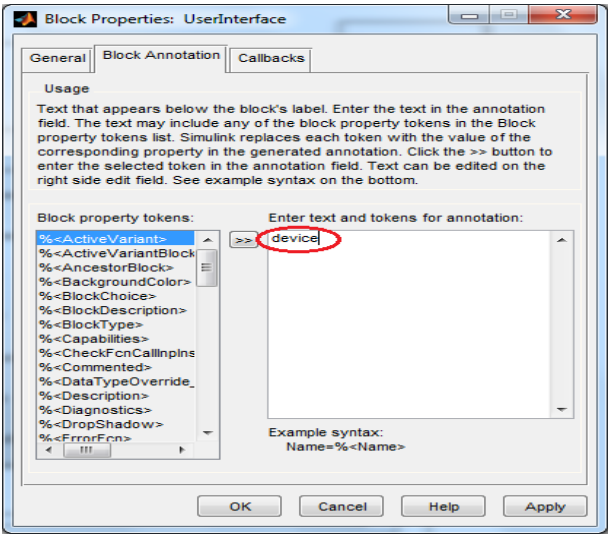

Figura 27 – Caixa de Diálogo do Simulink: *Block Properties*

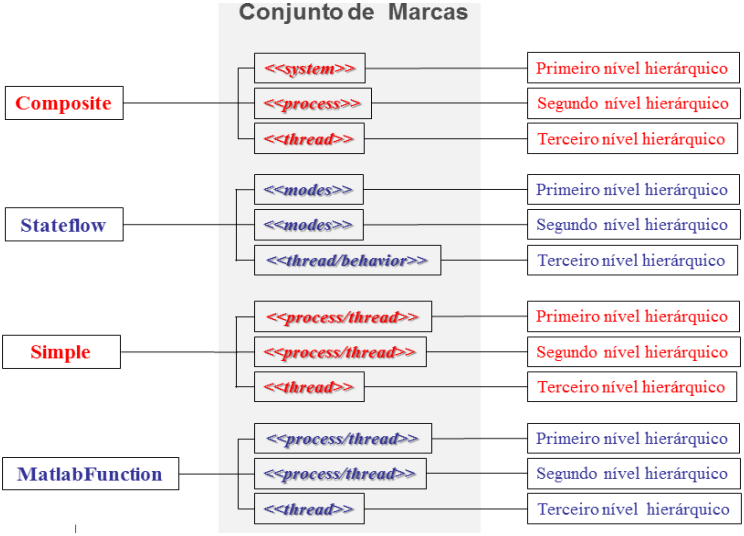

Figura 28 – Conjunto de Marcas - Marcação Automática.

diferenças entre a marcação manual e a marcação automática são as variações de mapeamento. Como pode ser observado na Figura 26, a

marcação manual possibilita diferentes variações do mapeamento para um mesmo tipo e posição hierárquica de bloco *Subsystem*, enquanto que a marcação automática, conforme Figura 28, possui uma única opção de marcação para cada tipo e posição hierárquica de bloco *Subystem*.

5.5 REGRAS DE TRANSFORMAÇÃO

Esta seção apresenta as regras de transformação do processo de transformação de modelos proposto pela AST para transformar modelos funcionais Simulink em modelos de arquitetura AADL. As regras de transformação da AST são unidirecionais, ou seja, elas especificam como ocorre a transformação de um componente do modelo funcional Simulink em um componente de software do modelo de arquitetura em AADL.

Depois de analisar as características dos metamodelos, e levando sempre em consideração a *correlação âncora* apresentada anteriormente, foram definidas as regras de transformação da AST. As regras de transformação foram especificadas em nível de metamodelo, de tal forma que elas são aplicáveis a todo o conjunto de modelos fonte que estejam em conformidade com o metamodelo dado. Elas são guiadas diretamente pelas marcas inseridas nos blocos *Subsystem* do modelo funcional Simulink, podendo ser classificadas em três categorias: mapeamento estrutural, mapeamento dos modos de operação e mapeamento comportamental.

Durante a execução do mapeamento estrutural todos os blocos *Subsystem* marcados no modelo funcional Simulink geram pelo menos um componente de software no modelo AADL com a mesma interface (portas). Por outro lado, o mapeamento dos modos de operação e o mapeamento comportamental não geram componentes no modelo AADL, eles geram seções na especificação de determinados componentes de software que foram gerados pelo mapeamento estrutural. Isso acontece porque eles manipulam diretamente blocos *Stateflow*. O mapeamento dos componentes (estados, transições, condições e ações) do diagrama *Stateflow* correspondente, nos dois casos, é sempre de um para um. O mapeamento dos modos de operação é executado exclusivamente quando um bloco *Stateflow* está posicionado no segundo nível hierárquico do modelo funcional Simulink, ou então, no mesmo nível dos processos. Já o mapeamento comportamental é executado somente quando o bloco *Stateflow* está posicionado no último nível hierárquico do modelo funcional Simulink.

Segundo (SAQUI-SANNES; HUGUES et al., 2012), o mapeamento de um para um além de garantir uma fronteira rigorosa entre dois domínios distintos, torna possível a implementação futura de um mapeamento reverso, além de ser um recurso altamente benéfico para assegurar a rastreabilidade dos componentes.

A Figura 29 apresenta as possíveis variações de mapeamento para os tipos de blocos *Subsystem* de um modelo funcional Simulink.

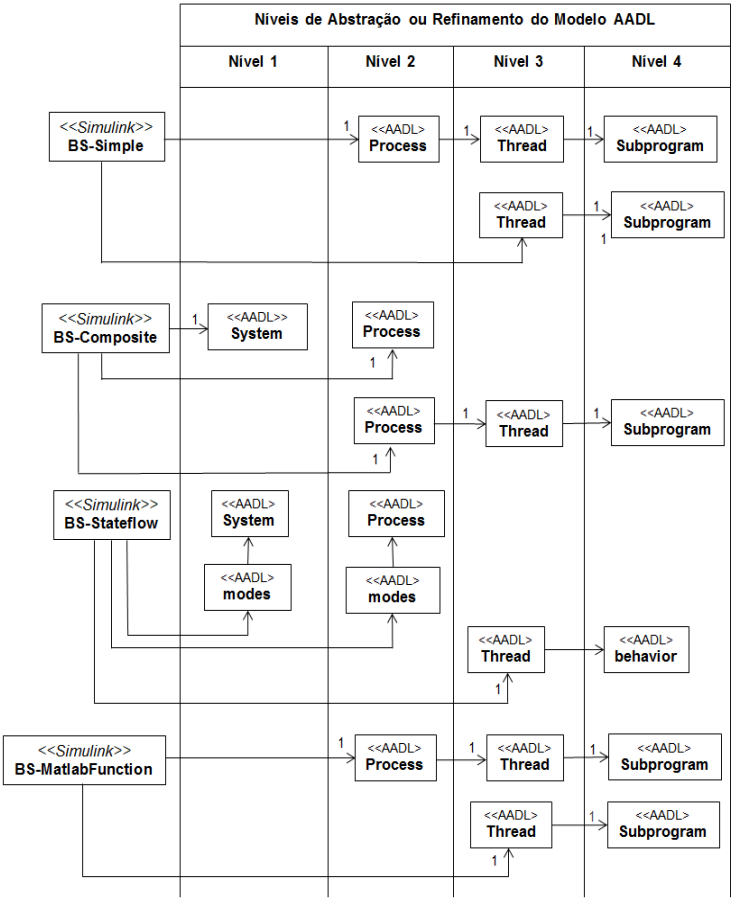

Figura 29 – Variações de Mapeamento

De acordo com a Figura 29, um bloco *Subsystem* do tipo *Simple* (BS-Simple) pode ser mapeado no segundo nível hierárquico do modelo de arquitetura AADL como um componente de software do tipo

Process, que possui no terceiro nível hierárquico um componente de software do tipo *Thread*, que por sua vez possui no quarto nível hierárquico um componente *Subprogram*. Outra variação de mapeamento para um bloco *Subsystem* do tipo *Simple* (BS-Simple), é que ele pode ser mapeado no terceiro nível hierárquico do modelo de arquitetura AADL como um componente de software do tipo *Thread* que possui no quarto nível hierárquico um componente *Subprogram*. As variações de mapeamento para os blocos *Subsystem* dos tipos *Composite, Stateflow* e *MatlabFunction* podem ser interpretadas seguindo a lógica das variações de mapeamento do bloco *Subsystem* do tipo *Simple* descrita anteriormente.

As variações de mapeamento estão diretamente relacionadas com as regras de transformação da AST. O que define a variação de mapeamento e consequentemente a regra de transformação que vai ser executada pelo motor de transformação da AST é a marca que o bloco *Subsystem* possui.

5.5.1 Mapeamento Estrutural

O principal objetivo do mapeamento estrutural da AST é extrair aspectos arquiteturais do modelo funcional Simulink para gerar um modelo de arquitetura de software preliminar em AADL. O mapeamento estrutural da AST possui as seguintes regras de transformação:

- **Simple-to-Process/Thread**. Um bloco *Subsystem* do tipo *Simple* que recebeu a marca *process/thread* é mapeado como um componente AADL do tipo *Process* que encapsula uma *Thread* que chama um *subprogram* - três componentes AADL são gerados neste caso.

- **Simple-to-Thread**. Um bloco *Subsystem* do tipo *Simple* que recebeu a marca *thread* é mapeado como uma *Thread* que chama um *subprogram* - dois componentes AADL são gerados neste caso.

- **Simple-to-Device**. Um bloco *Subsystem* do tipo *Simple* com a marca *Device* é mapeado como um componente AADL do tipo *Device*.

- **Composite-to-System**. Um bloco *Subsystem* do tipo *Composite* que recebeu a marca *system* é mapeado como um componente AADL do tipo *System*.

- **Composite-to-Process**. Um bloco *Subsystem* do tipo *Composite* que recebeu a marca *process* é mapeado como um componente AADL do tipo *Process*.

- **Composite-to-Device**. Um bloco *Subsystem* do tipo *Composite* com a marca *Device* é mapeado como um componente AADL do tipo *Device*.

- **MATLABFunction-to-Process/Thread**. Um bloco *Subsystem* do tipo *MatlabFunction* que recebeu a marca *process/thread* é mapeado como um componente AADL do tipo *Process* que encapsula uma *Thread* que chama um *subprogram* - três componentes AADL são gerados neste caso.

- **MATLABFunction-to-Thread**. Um bloco *Subsystem* do tipo *MatlabFunction* que recebeu a marca *thread* é mapeado como uma *Thread* que chama um *subprogram* - dois componentes AADL são gerados neste caso.

As portas dos tipos *signal name*, *port number* e *port number and signal name* de um bloco *Subsystem* do modelo funcional Simulink são mapeadas respectivamente como portas do tipo *event port*, *data port* e *event data port* dos respectivos componentes no modelo AADL. Os tipos de dados suportados pelas portas do bloco *Subsystem* são mapeados como sendo o tipo de dado suportado pelas respectivas portas do tipo *data port* e *event data port* do componente no modelo AADL. O mapeamento estrutural gera no modelo AADL um pacote chamado *dados* com os tipos de dados que trafegam pelas portas dos blocos *Subsystem* no modelo funcional Simulink, e também um pacote chamado *subprograms* com especificações básicas dos subprogramas chamados pelas *threads* no modelo AADL. As linhas que conectam as portas dos blocos *Subsystem* no modelo funcional Simulink são mapeadas como conexões entre seus respectivos componentes no modelo AADL.

Visando um melhor entendimento do papel das marcas e da execução do mapeamento estrutural da AST, observe o exemplo gráfico apresentado na Figura 30, a qual apresenta o primeiro nível hierárquico do modelo funcional Simulink de um sistema embarcado que realiza manobras de estacionamento de forma autônoma (SIAMES). Ele é responsável por controlar a velocidade do veículo, por encontrar uma vaga de estacionamento compatível com o veículo e por controlar a manobra de estacionamento propriamente dita.

O primeiro nível hierárquico do modelo funcional Simulink é o ponto de partida do processo de transformação de modelos da AST. Neste nível o modelo é composto por quatro blocos *subsystem*, um que representa o sistema propriamente dito *(SystemParking)*, que é do tipo *Composite*, outro que representa a interação do sistema com o mundo exterior *(UserInterface)*, que é do tipo *Simple*. Os blocos *SensorSlot* e *SensorSpeed* são blocos *subsystem* do tipo *Simple* e representam dois sensores que enviam sinais para o sistema. Os blocos *Subsystem* foram marcados manualmente utilizando o recurso de anotação de blocos da própria ferramenta Simulink, e principalmente, levando em consideração a posição hierárquica de cada bloco *Subsystem* dentro do modelo funcional Simulink. As marcas aparecem na parte inferior do blocos.

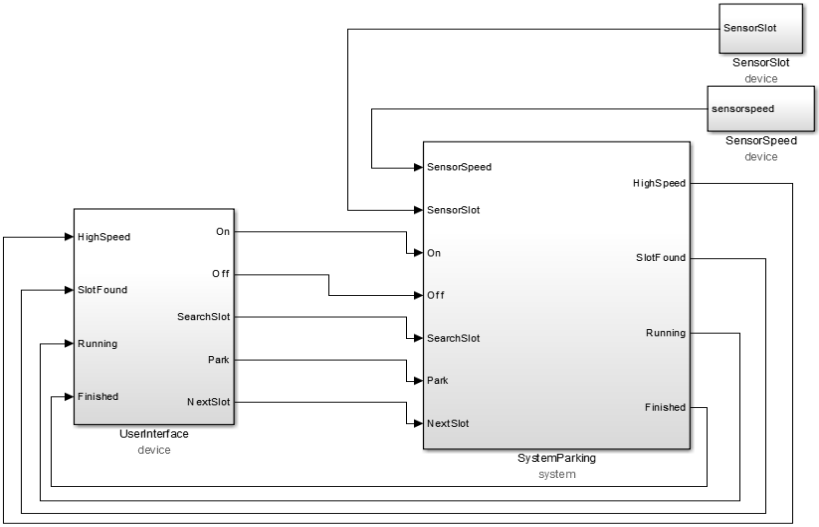

Figura 30 – Modelo Funcional Simulink de Nível 1- SIAMES

O modelo AADL correspondente de nível 1 gerado pelo mapeamento estrutural da AST é mostrado na Figura 31. Note que na linha 1 este modelo contém um componente *system* raíz chamado *s_siames.impl*, que possui como subcomponentes um componente *system* chamado *s_systemparking* e outros três componentes *devices* identificados como *d_userinterface, d_sensorspeed* e *d_sensorslot* (linhas 3 a 6). Todos os componentes foram mapeados levando em consideração suas respectivas marcas. As portas e as linhas dos blocos do modelo

```
1   SYSTEM IMPLEMENTATION s_siames.impl
2   subcomponents
3     d_userinterface: device d_userinterface.impl;
4     s_systemparking: system s_systemparking.impl;
5     d_sensorspeed: device d_sensorspeed.impl;
6     d_ssensorslot: device d_sensorslot.impl;
7   connections
8      port userinterface.on -> systemparking.on;
9      port userinterface.off -> systemparking.off;
10     port userinterface.searchslot -> systemparking.searchslot;
11     port userinterface.park -> systemparking.park;
12     port userinterface.nextslot -> systemparking.nextslot;
13     port systemparking.highspeed -> userinterface.highspeed;
14     port systemparking.slotfound -> userinterface.slotfound;
15     port systemparking.running -> userinterface.running;
16     port systemparking.finished -> userinterface.finished;
17     port sensorspeed.sensorspeed ->systemparking.sensorspeed;
18     port sensorslot.sensorslot -> systemparking.sensorslot;
19  END siames.impl;
```

Figura 31 – Modelo AADL de Nível 1 - SIAMES)

funcional Simulink foram mapeadas automaticamente como portas e conexões no modelo AADL (linhas 7 a 18).

Outra perspectiva do mapeamento estrutural entre os modelos Simulink e AADL é apresentado na Figura 32. No lado esquerdo aparece um trecho do arquivo .mdl do modelo funcional Simulink do sistema SIAMES e do lado direito aparece o trecho correspondente do arquivo .aadl gerado após a execução do mapeamento estrutural. Os trechos destacados em vermelho mostram que o bloco *Subsystem* chamado *SearchingSlot* que recebeu a marca *Process* gerou no modelo AADL um componente de software do tipo *Process* chamado *SearchingSlot*. Os trechos em azul mostram que os blocos do tipo *Inport* e *Outport* que pertencem ao bloco *Subsystem SearchingSlot* no arquivo .mdl foram mapeado no modelo AADL como portas de entrada e saída do componente *Process SearchingSlot* no modelo AADL. Os trechos em verde mostram que as linhas do bloco *Subsystem SerachingSlot* foram mapeadas como conexões do componente *Process SearchingSlot* no modelo AADL.

5.5.2 Mapeamento dos Modos de Operação

Como o nome sugere, o objetivo deste mapeamento é obter informações sobre os possíveis modos de operação do modelo funcional Simulink e mapear para o modelo de arquitetura AADL. Portanto, é necessário o uso de informações dos blocos *Subsystem* do tipo *Stateflow*, que podem expressar Máquinas de Estado - nomeados em Simulink como Diagramas *Stateflow*. Em AADL os modos de operação estão associados a componentes e são representados por uma máquina de

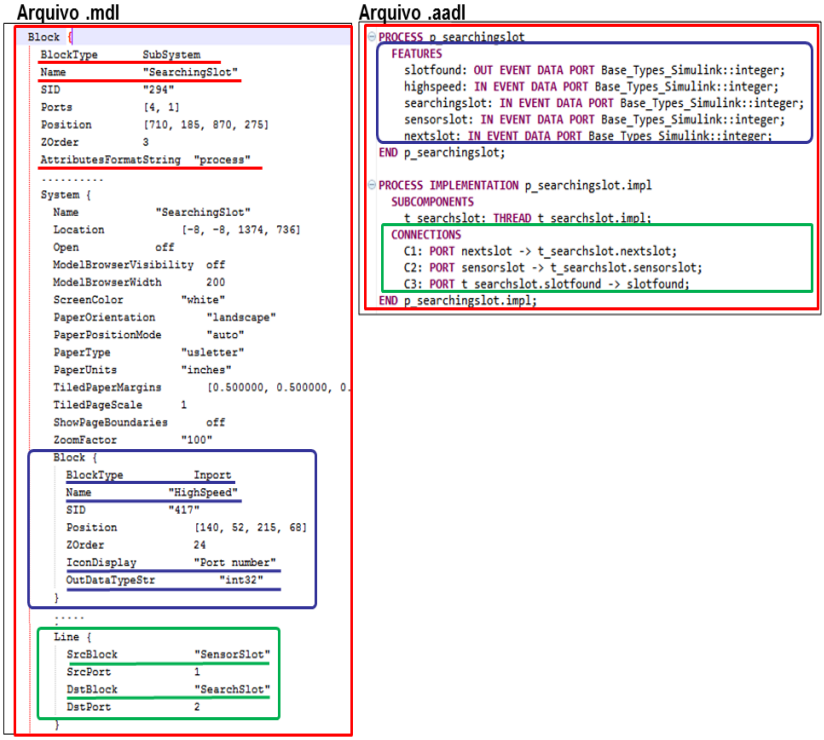

Figura 32 – Exemplo da Execução do Mapeamento Estrutural

estados modal. Esse mapeamento possui a seguinte regra de transformação:

- **Stateflow-to-OperationMode**. Um bloco *Subsystem* do tipo *Stateflow* que recebeu a marca *modes* afeta o seu bloco *Subsystem* pai (o bloco no nível hierárquico imediatamente superior). O componente AADL correspondente (gerado de acordo com o mapeamento estrutural) recebe os modos de operação. Os estados do diagrama *Stateflow* representam os modos de operação, as transições entre os estados representam as transições entre os modos de operação, e as ações dos estados representam a geração dos eventos que desencadeiam a ativação dos processos AADL relacionados.

Em linhas gerais, o mapeamento dos modos de operação mapeia apenas as portas de entrada de um diagrama *Stateflow*, isso porque são elas que disparam as transições entre os modos de operação do sistema. O mapeamento dos modos de operação da AST identifica os processos que devem estar ativos em cada modo de operação do sistema a partir das ações especificadas nos estados do diagrama *Stateflow* que está sendo mapeado, e adiciona a declaração ***in modes*** nos respectivos processos do sistema no modelo AADL.

Dando continuidade ao exemplo, apresenta-se na Figura 33 o segundo nível hierárquico do modelo funcional Simulink do sistema SIAMES, modelo este correspondente ao refinamento do bloco *SystemParking* da Figura 30. Neste nível hierárquico, o modelo funcional Simulink possui quatro blocos *Subsystem*, sendo um do tipo *Stateflow* identificado como *OperationModes*, outros dois do tipo *Composite* identificados como *SpeedController*, *SearchingSlot*, e um outro do tipo *Simple* identificado como *ParkingManouver*. Conforme informado anteriormente, a marca que cada bloco *Subsystem* recebeu aparece na parte inferior dos mesmos.

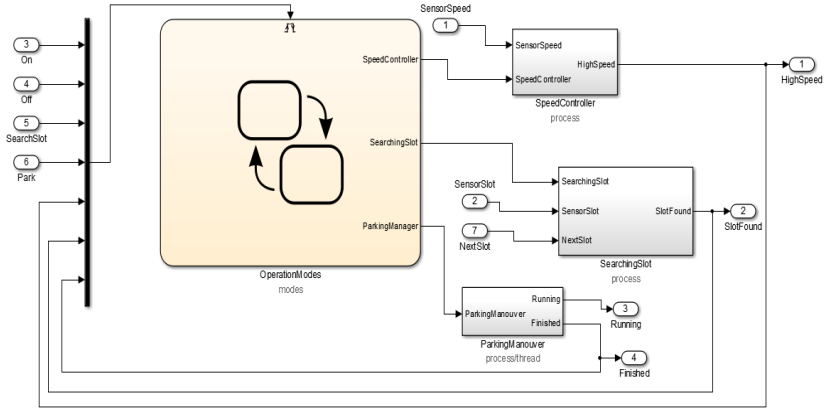

Figura 33 – Modelo Funcional de Nível 2 - Refinamento do Bloco *SystemPark* - SIAMES

O diagrama *Stateflow* do bloco *OperationModes* é apresentado na Figura 34. Ele é formado por três estados básicos identificados como *Idle, Lookup e Park*, os quais representam os modos de operação do sistema, e possui um conjunto de transições de estado ativadas exclusivamente por eventos. Observe que os estados *Lookup* e *Park* possuem ações de estado do tipo *during*. Estas ações de estado ativam

os blocos *SpeedController* e *SearchingSlot* enquanto o sistema estiver com o modo de operação *Lookup* ativo, e os blocos *SpeedController* e *ParkingManouver* enquanto o sistema estiver com o modo de operação *Park* ativo.

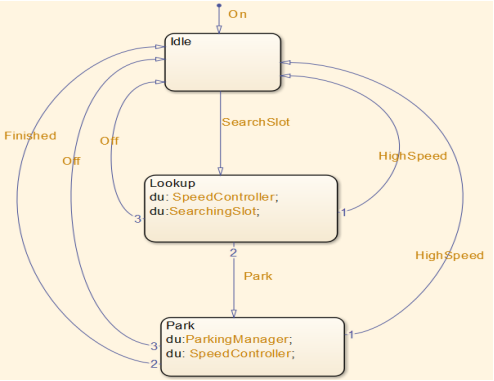

Figura 34 – Diagrama Stateflow do bloco Stateflow ModosDeOperação

O modelo AADL correspondente gerado pela AST é apresentado na Figura 35. Neste caso, o mapeamento estrutural gerou três componentes de software do tipo *Process* identificados como *p_speedcontroller*, *p_searchingslot* e *p_parkingmanouver* dentro do componente *systemparking* do modelo AADL de nível 2, veja linhas 3 à 6. Mesmo suprimido da Figura 35, o processo *p_parkingmanouver* possui internamente um componente de software do tipo *thread* que chama um componente *subprogram*, isso aconteceu porque o bloco *Subsystem ParkingManouver* recebeu a marca *process_thread* no modelo funcional Simulink de nível 2.

Considerando que o bloco *Stateflow* recebeu a marca *modes*, e que isto aconteceu porque ele está posicionado no segundo nível hierárquico do modelo funcional Simulink, o mapeamento dos modos de operação da AST gerou a máquina de estados modal do componente *s_systemparking* do modelo AADL. Ou seja, o componente pai do bloco *Stateflow*, veja linhas 18 a 35. Baseados nas ações de estado especificadas no diagrama *Stateflow* da figura 34, os processos são ativados de acordo modo de operação ativo do sistema. Em AADL, o comando *in modes* é utilizado para especificar em que modo de operação o processo deve ser ativado.

A Figura 36 apresenta outra perspectiva do mapeamento dos modos de operação do modelo funcional Simulink para o modelo de

```
1   SYSTEM IMPLEMENTATION s_systemparking.impl
2   SUBCOMPONENTS
3       p_speedcontroller : PROCESS p_speedcontroller.impl
4       in modes (m_lookup,m_park);
5       p_searchingslot : PROCESS p_searchingslot.impl in modes (m_lookup);
6       p_parkingmanouver : PROCESS p_parkingmanouver.impl in modes (m_park);
7   CONNECTIONS
8       PORT sensorspeed -> p_speedcontroller.sensorspeed;
10      PORT p_searchingslot.slotfound -> slotfound;
11      PORT sensorslot -> p_searchingslot.sensorslot;
12      PORT nextslot -> p_searchingslot.nextslot;
13      PORT p_parkingmanouver.running -> running;
14      PORT p_speedcontroller.highspeed -> highspeed;
15      p_speedcontroller.highspeed -> p_searchingslot.highspeed;
17      PORT p_parkingmanouver.finished -> finished;
18  modes
19      m_idle: initial mode;
20      m_lookup: mode;
21      m_park: mode;
22
23      m_idle: initial mode;
24      m_lookup: mode;
25      m_park: mode;
26      m_idle -[on]-> m_idle;
27      m_park -[p_parkingmanouver.finished]-> m_idle;
28      m_park -[p_searchingslot.highspeed]-> m_idle;
29      m_park -[p_speedcontroller.highspeed]-> m_idle;
30      m_park -[off]-> m_idle;
31      m_lookup-[p_searchingslot.highspeed]-> m_idle;
32      m_lookup-[p_speedcontroller.highspeed]-> m_idle;
33      m_lookup-[off]-> m_idle;
34      m_idle -[searchslot]-> m_lookup;
35      m_lookup-[park]-> m_park;
36  END s_systemparking.impl;
```

Figura 35 – Modelo AADL de Nível 2 - Refinamento do componente *system s_systemparking.impl* - SIAMES

arquitetura AADL. Os trechos em vermelho mostram que o bloco do tipo *Subsystem* chamado *SystemParking* no modelo funcional Simulink (arquivo .mdl) foi mapeado no modelo AADL (arquivo .aadl) como um componente do tipo *System* chamado *s_systemParking.impl*. Os trechos em azul mostram que o bloco *Subsystem* chamado *OperationModes* que possui a marca *Modes* e que pertence ao bloco *SystemParking* no arquivo .mdl gerou no arquivo .aadl a seção *MODES*, a qual especifica os modos de operação do componente *s_systemparking.impl* do modelo AADL. Todo bloco *Subsystem* do tipo *Stateflow* no modelo Simulink (arquivo .mdl) possui associado a ele um *Chart*, como pode ser observado na Figura 36. Um bloco *Subsystem* é classificado como sendo do tipo *Stateflow* quando ele possuir a propriedade *SFBlock Type* definida como *"Chart"*. Os trechos em verde mostram que os estados do *Chart OperationModes* foram mapeados como sendo os modos de operação do sistema no modelo AADL. Já os trechos em laranja demonstram que as transições do *Chart OperationModes* foram mapeadas como sendo as transições entre os modos de operação no modelo AADL.

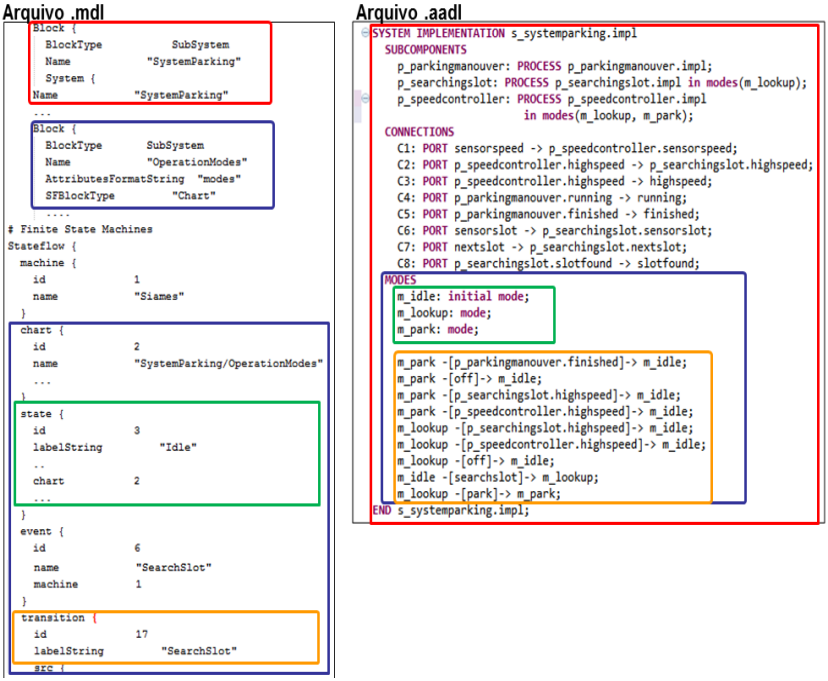

Figura 36 – Exemplo da Execução do Mapeamento dos Modos de Operação

5.5.3 Mapeamento Comportamental

O principal objetivo do mapeamento comportamental é extrair os aspectos comportamentais de blocos *Subsystem* do modelo funcional Simulink e mapear para componentes do modelo de arquitetura AADL. Como acontece no mapeamento dos modos de operação, o mapeamento comportamental também manipula os blocos *Stateflow*. No entanto, neste caso, eles devem estar posicionados dentro de um bloco *Subsystem* que recebeu a marca *process*.

O resultado deste mapeamento é a geração de um componente de software AADL do tipo *Thread*, com seu comportamento interno devidamente especificado. Em AADL, a especificação do comportamento interno de um componente é representado por uma espécie de máquina de estados com condições de guarda e ações, conforme definição do anexo comportamental da AADL (SAE, 2011a). A seção

annex behavior_specification, no caso do mapeamento comportamental, é inserida no respectivo componente *Thread*. Segue o conjunto de transformações realizadas neste nível.

- **Stateflow-to-thread/behavior**. O bloco *Subsystem* do tipo *Stateflow* que recebeu a marca *thread/behavior* é mapeado estruturalmente como uma *Thread* no modelo AADL. Essa *Thread* recebe uma seção chamada *annex behavior specification* gerada a partir do diagrama *Stateflow*. A seção *annex behavior_specification* especifica o comportamento interno da *Thread*, isto inclui estados, transições, condições de guarda e ações. Diferentemente das ações que ocorrem durante a especificação dos modos de operação, aqui as ações representam especificações definidas pelo usuário que ocorrem quando o estado de destino é alcançado.

Finalizando o exemplo gráfico, observe a Figura 37. Esta figura apresenta o refinamento do bloco *Subsystem SpeedController* da Figura 33, o qual possui internamente um bloco *Stateflow* denominado *SpeedController1*. O bloco *Stateflow SpeedController1* possui um diagrama *Stateflow* que está posicionado no último nível hierárquico do modelo funcional Simulink. Este diagrama *Stateflow* é apresentado na Figura 38, ele é composto por três estados básicos identificados como *Idle, NormalSpeed e HighSpeed*, os quais representam o comportamento da função responsável pelo monitoramento de velocidade do sistema SIAMES, e possui também um conjunto de transições de estado que são disparadas de acordo com chegada de eventos e/ou dados pelas portas bloco *Stateflow*.

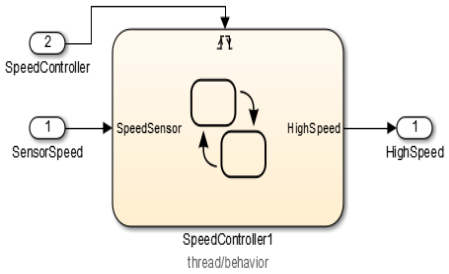

Figura 37 – Modelo Funcional de Nível 3 - Refinamento do Bloco *SpeedController* - SIAMES

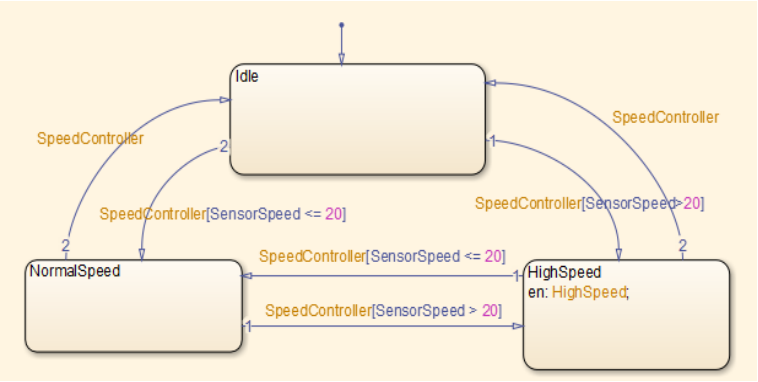

Figura 38 – Diagrama Stateflow do bloco Stateflow *SpeedController1*

O modelo AADL correspondente gerado pela AST é mostrado na Figura 39. Neste caso, o mapeamento comportamental gerou a partir da marca *thread/behavior* um componente AADL do tipo *thread* chamado *t_speedcontroller.impl*, que por sua vez é um subcomponente do processo *p_speedcontroller*, veja linhas 9 e 10. Vale frisar, que o processo *p_speedcontroller* foi criado anteriormente a partir do mapeamento do modelo funcional Simulink de nível 2 (veja Figura 33).

O mapeamento comportamental da AST, baseado no diagrama *Stateflow* do bloco *Stateflow* gerou o comportamento interno da *thread* *t_speedcontroller* (veja linhas de 25 a 54). Neste caso, não existe a chamada de um *subprogram*, isso acontece porque o comportamento da *thread* é especificado pela seção *annex behavior_specification*, estes recursos são mutuamente exclusivos.

A Figura 40 apresenta outra perspectiva do mapeamento comportamental do modelo funcional Simulink para o modelo de arquitetura AADL. Os trechos em vermelho mostram que o bloco do tipo *Subsystem* chamado *SpeedController1* que possui a marca *thread/behavior* no arquivo .mdl gerou no modelo AADL como um componente de software do tipo *Thread* chamado *t_speedController1* que possui a seção *annex behavior_specification* definida. Os trechos em azul mostram que os estados do *Chart SpeedController1* foram mapeados como sendo os estados do comportamento da *Thread t_speedController1* no modelo AADL. Os trechos em verde demonstram que as transições do *Chart SpeedController1* do arquivo .mdl foram mapeadas como sendo as transições entre os estados do comportamento da *Thread t_speedController1* no modelo AADL. Já os trechos em laranja demonstram que os dados

```
1  PROCESS p_speedcontroller
2  FEATURES
3    sensorspeed : IN EVENT DATA PORT base_types::integer;
4    speedcontroller : IN EVENT PORT ;
5    highspeed : OUT EVENT PORT ;
6  END p_speedcontroller;
7
8  PROCESS IMPLEMENTATION p_speedcontroller.impl
9  SUBCOMPONENTS
10   t_speedcontroller : THREAD t_speedcontroller.impl;
11 CONNECTIONS
12   c1 : PORT sensorspeed -> t_speedcontroller.sensorspeed;
13   c2 : PORT speedcontroller -> t_speedcontroller.speedcontroller;
14   c3 : PORT t_speedcontroller.highspeed -> highspeed;
15 END p_speedcontroller.impl;
16
17 THREAD t_speedcontroller
18 FEATURES
19   sensorspeed : IN EVENT DATA PORT base_types::integer ;
20   speedcontroller : IN EVENT PORT;
21   highspeed : OUT EVENT PORT;
22 END t_speedcontroller;
23
24 THREAD IMPLEMENTATION t_speedcontroller.impl
25 subcomponents
26     dt_sensorspeed: data Base_Types::integer;
27 annex behavior_specification {**
28  states
29      s_idle : initial complete state;
30      s_intermediate_normalspeed: state;
31      s_normalspeed :  complete state;
32      s_intermediate_highspeed: state;
33      s_highspeed :   complete state;
34
35   transitions
36      s_idle -[on dispatch speedcontroller]-> s_intermediate_normalspeed
37      {sensorspeed?(dt_sensorspeed)};
38      s_intermediate_normalspeed -[dt_sensorspeed <= 20]-> s_normalspeed;
39      s_normalspeed -[on dispatch speedcontroller]-> s_idle;
40
41      s_idle -[on dispatch speedcontroller]-> s_intermediate_highspeed
42      {sensorspeed?(dt_sensorspeed)};
43      s_intermediate_highspeed -[dt_sensorspeed > 20]-> s_highspeed{highspeed!};
44      s_highspeed -[on dispatch speedcontroller]-> s_idle;
45
46      s_highspeed -[on dispatch sensorspeed]-> s_intermediate_highspeed
47      {sensorspeed?(dt_sensorspeed)};
48      s_intermediate_highspeed -[dt_sensorspeed <= 20 ]-> s_normalspeed;
49
50      s_normalspeed -[on dispatch sensorspeed]-> s_intermediate_normalspeed
51      {sensorspeed?(dt_sensorspeed)};
54      s_intermediate_normalspeed -[dt_sensorspeed > 20 ]-> s_highspeed{highspeed!};
54 **};
55 END t_speedcontroller.impl;
```

Figura 39 – Modelo AADL de Nível 3 - Comportamento da thread *t_speedcontroller.impl* - SIAMES

do *Chart SpeedController1* foram mapeados como dados que são manipulados pelo comportamento da *Thread t_speedcontroller1* no modelo AADL.

5.6 PLUGIN AS2T

O *plugin* AS2T (Tradutor Simulink/AADL) foi implementado no escopo deste trabalho de pesquisa como uma prova de conceito para verificar a execução dos mapeamentos propostos pela AST quando

Figura 40 – Exemplo da Execução do Mapeamento Comportamental

automatizados. De acordo com a Figura 41, o *plugin* AS2T gera um modelo AADL textual a partir de informações extraídas de um modelo funcional Simulink importado. O *plugin* AS2T foi implementado usando a linguagem de programação Java e empacotado como um *plugin* para o ambiente OSATE (*Open Source Architectural Environment Tool*).

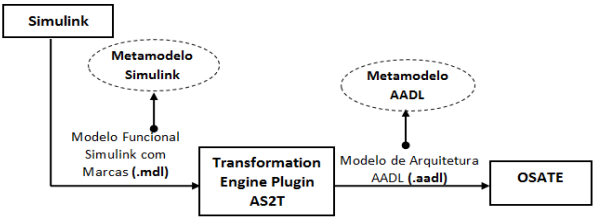

Figura 41 – Estrutura do Plugin AS2T

O *plugin* AS2T possui um método *parser* que lê linha a linha o arquivo *.mdl* importado, identifica e gera uma lista encadeada contendo todos os elementos pai e os respectivos elementos filhos de cada bloco *Subsystem* existente no modelo funcional Simulink importado. Uma variável chamada *mdlFile* gerada pelo *parser* possui apenas as propriedades dos blocos *Subsystem* relevantes para a execução das regras de transformação da AST, isso foi feito para tornar a implementação das regras de transformação mais simples, e, além disso, proporcionar ganho no tempo de execução das mesmas.

O *plugin* AS2T possui um método *autoMark*, conhecido como assistente de marcação, ele tem a responsabilidade de inserir as marcas nos blocos *Subsystem* da variável *mdlFile*, caso estes não possuam marcas previamente inseridas pelo projetista. Como já discutido, as marcas são inseridas de acordo com o tipo do bloco *Subsystem* (*Composite, Simple, Stateflow ou MATLABFunction*) e de acordo com a posição hierárquica do mesmo dentro do arquivo mdl.

Depois da execução dos métodos *parser* e do *autoMark* o motor de transformação (*transformEngine*) executa as regras de transformação da AST, gera o objeto AADL e posteriormente escreve o modelo AADL propriamente dito. A Figura 42 apresenta a regra de transformação que é executada quando a marca *process/thread* é encontrada em algum bloco *Subsystem* da variável *mdlFile*. Neste caso em específico, é gerado no modelo AADL um componente de software do tipo *Process* que possui internamente um componente de software do tipo *Thread* com as mesmas portas de entrada e saída do componente *Process* pai.

```
730     if(this.mark.equals(MARK_PROCESS_THREAD)){
731         this.addSubSystem(new SubSystem(this.getName(), aadlFile, this));
732         this.getLastSubSystem().setMark(MARK_THREAD);
733         this.getLastSubSystem().inPorts = this.inPorts;
734         this.getLastSubSystem().outPorts = this.outPorts;
735         for (int i = 0; i < this.getLastSubSystem().getInPorts().size(); i++) {
736             if(this.getLastSubSystem().name.equals(this.getLastSubSystem().getInPorts().get(i).getName())){
737                 addLine(this.getLastSubSystem().getInPorts().get(i).getName(), "0", this.getLastSubSystem().getName(), "" + (i + 1));
738             }else{
739                 addLine(this.getLastSubSystem().getInPorts().get(i).getName(), "1", this.getLastSubSystem().getName(), "" + (i + 1));
740             }
741         }
742         for (int i = 0; i < this.getLastSubSystem().getOutPorts().size(); i++) {
743             this.addLine(this.getLastSubSystem().getName(), "" + (i + 1), this.getLastSubSystem().getOutPorts().get(i).getName(), "1");
744         }
745     }
```

Figura 42 – Regra de transformação ativada pela marca *process/thread*

A Figura 43 apresenta a regra de transformação que é executada quando a marca *thread/behavior* é encontrada em algum bloco *Subsystem* da variável *mdlFile*. Esta regra mapeia o bloco *Subsystem* com tal marca como um componente de software do tipo *Thread* com uma seção *annex behavior specification* no modelo AADL. A seção *annex behavior specificacion* possui os estados e as transições do diagrama Stateflow

associado ao bloco *Subsystem* que recebeu a marca *thread/behavior* no modelo funcional Simulink.

```
892 if(this.mark.equals(MARK_THREAD_BEHAVIOR)){
893     if(this.getBehavior() != null){
894         ret += "  annex behavior_specification {**\n";
895         ret += "    STATES\n";
896         for (int i = 0; i < this.getBehavior().getModes().size(); i++) {
897             ret += "      " + this.getBehavior().getModes().get(i).getFullName() + ": " +
898                     (this.getBehavior().getModes().get(i).isInitial() ? "initial " : "") +
899                     (this.getBehavior().getModes().get(i).isComplete() ? "complete " : "") + "state;\n";
900         }
901         if(this.getBehavior().getTransitions().size() > 0){
902             ret += "    TRANSITIONS\n";
903             for (int i = 0; i < this.getBehavior().getTransitions().size(); i++) {
904                 if(this.getBehavior().getTransitions().get(i).getOrigin() != null){
905                     ret += "      " + this.getBehavior().getTransitions().get(i).getOrigin().getFullName() + " -";
906                     ret += "[" + this.getBehavior().getTransitions().get(i).getLabel() + "]-> ";
907                     ret += this.getBehavior().getTransitions().get(i).getDestiny().getFullName();
908                     if(!this.getBehavior().getMode(this.getBehavior().getTransitions().get(i).getDestiny().getId()).getAnnotation().equals("")){
909                         ret += " " + this.getBehavior().getMode(this.getBehavior().getTransitions().get(i).getDestiny().getId()).getAnnotation();
910                     }
911                     ret += ";\n";
912                 }
913             }
914         }
915         ret += "  **};\n";
916     }
917 }
```

Figura 43 – Regra de transformação correspondente a marca *thread/behavior*

Uma vez que o *plugin "as2t_v1.0.0.jar"* foi copiado para dentro da pasta *plugins* do OSATE, para utilizar o transformador de modelos AS2T basta instanciar o ambiente OSATE, criar um projeto AADL que armazenará o arquivo AADL gerado, clicar com o botão direito sobre o projeto AADL gerado e selecionar a opção *Import*. A caixa de diálogo que será aberta é apresentada na figura 44. Dentro da caixa de diálogo basta selecionar a opção *Importar Arquivo* da pasta *AS2T* conforme indicado na figura 44 e clicar no botão *Next*.

Na próxima caixa de diálogo é preciso selecionar o modelo funcional Simulink (arquivo .mdl) que será importado, a pasta onde ele ficará armazenado e clicar no botão *Finish*, conforme indicado na figura 45. É nesta mesma pasta que serão armazenados os arquivos .aadl gerados pelo transformador de modelos AS2T.

É possível observar na figura 46 que são adicionados a pasta selecionada na caixa de diálogo anterior (figura 45), o arquivo .mdl correspondente ao modelo funcional Simulink, o arquivo .aadl correspondente ao modelo de arquitetura de software AADL, o arquivo *Base_Types_Simulink.aadl* que armazena os tipos de dados que trafegam entre as portas do tipo *data port* e *event data port* do modelo AADL gerado, e o arquivo *Programs_Simulink.aadl* que armazena os *subprograms* chamados pelas *threads* no modelo AADL gerado.

A partir deste ponto, basta o projetista anotar o modelo AADL gerado com informações adicionais referentes a plataforma de execução

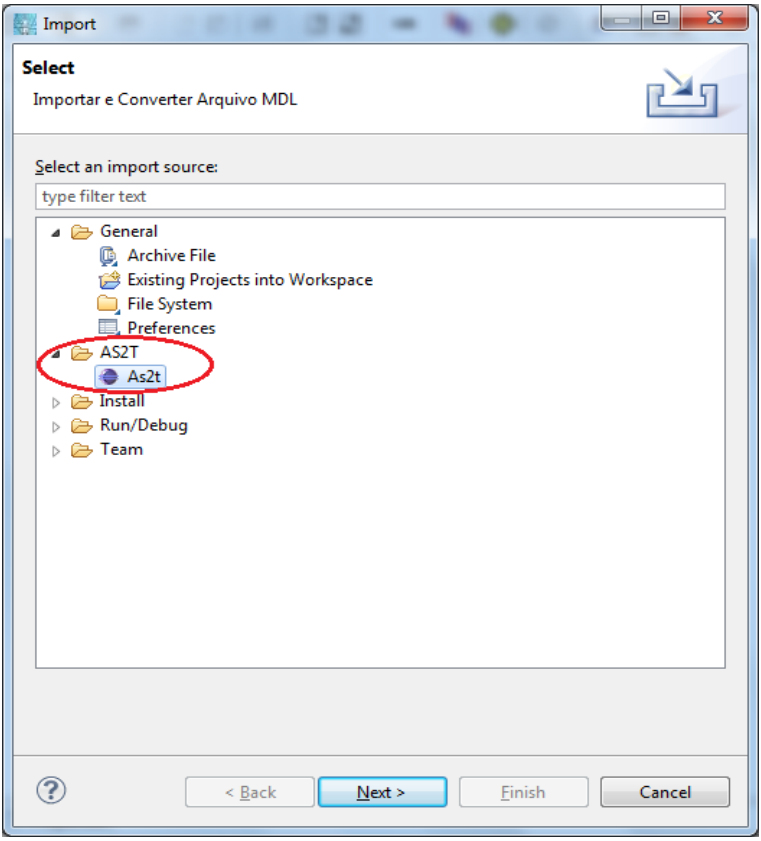

Figura 44 – Janela: Importar e Converter Arquivo MDL

e propriedades dos componentes de software, e submeter o modelo as análises e verificações desejadas.

Embora o modelo gerado pelo *plugin* AS2T seja considerado preliminar, grande parte do trabalho empregado para especificar os componentes de software do modelo de arquitetura do sistema é realizado de forma automática. Evitando, por exemplo, inconsistências na definição das conexões entre as portas dos componentes de software (*systems*,*devices* e *threads*), e garantindo a consistência dos dados entre as portas envolvidas nestas conexões. Outra contribuição interessante do *plugin* AS2T, que agiliza consideravelmente o trabalho do projetista, é que ele gera o sistema raiz que pode ser instanciado diretamente, composto por componentes *subgram*, chamadas de subprogramas dentro

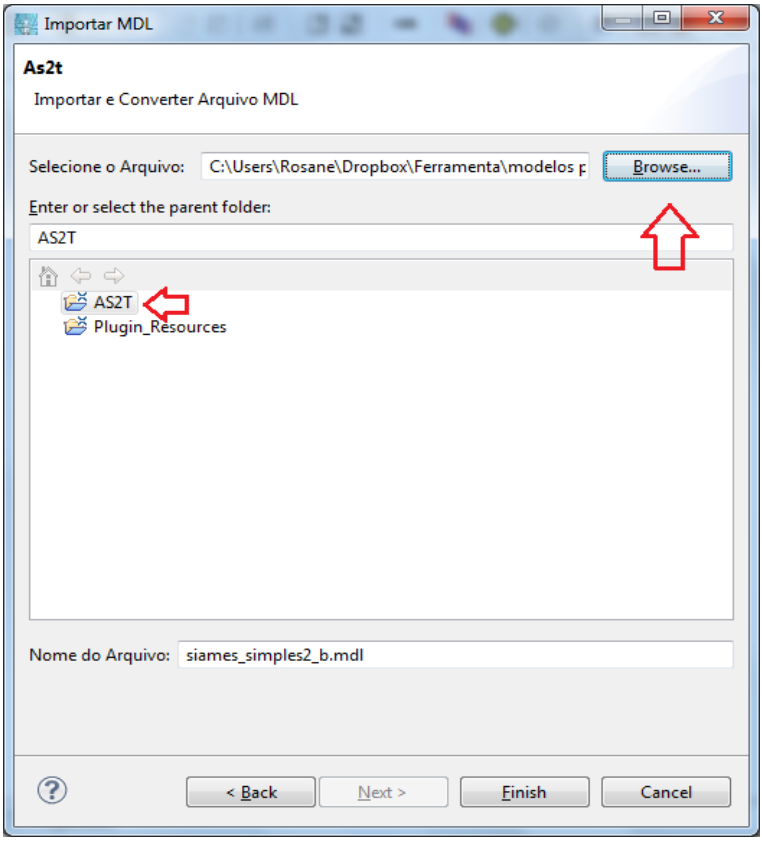

Figura 45 – Janela: Importar e Converter Arquivo MDL

das *threads*, e com os tipos de dados das portas do tipo *data port* e *event data port* devidamente especificados.

5.7 POSSÍVEIS AMEAÇAS RELACIONADAS A UTILIZAÇÃO DA AST

Por ser resultado de um trabalho de pesquisa experimental, a AST pode trazer possíveis problemas ao processo de desenvolvimento de um CPS.

Primeiramente, a AST somente poderá extrair as informações esperadas de um modelo funcional se este estiver corretamente estru-

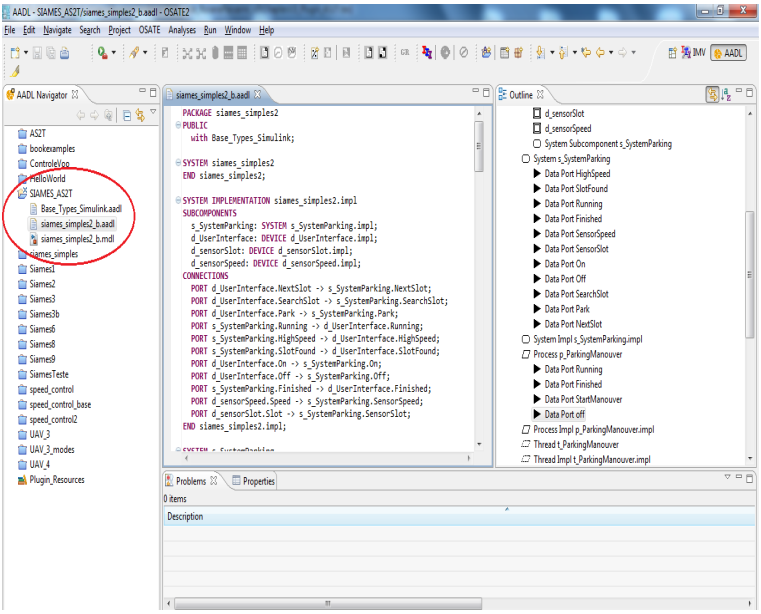

Figura 46 – Modelo de Arquitetura de Software Preliminar Gerado pelo Plugin AS2T

turado. Ou seja, o modelo funcional precisa expressar a hierarquia funcional do CPS. Uma hierarquia funcional deve expressar a hierarquia estrutural dos subsistemas que compõem o sistema de controle, suas interfaces e conexões. Sendo assim, a não aplicação das diretrizes de modelagem básicas apresentadas anterior por parte do usuário pode dificultar a identificação de aspectos estruturais no modelo funcional, e, como consequência, o modelo de arquitetura gerado pela AST pode não refletir de forma satisfatória a arquitetura SCE. O modelo de arquitetura vai ser gerado pela AST independentemente da aplicação das diretrizes de modelagem no modelo funcional importado, entretanto, neste caso, o modelo de arquitetura gerado pela AST pode requerer uma quantidade maior ajustes adicionais.

Um segundo ponto que pode gerar questionamentos, é o fato de que o motor de transformação de modelos da AST poder gerar modelos de arquitetura diferentes para um mesmo sistema, ou seja, a partir de um mesmo modelo funcional Simulink. Entretanto, vale lembrar que esta questão está relacionada com as variações de mapeamento suportadas pela AST, as quais por sua vez, estão diretamente relacionadas

com as marcas inseridas pelo usuário modelo no funcional Simulink. Sendo assim, o motor de transformação de modelos da AST vai gerar um modelo de arquitetura que reflete as decisões de projeto tomadas pelo usuário no momento da marcação do modelo funcional Simulink.

Para finalizar, as diretrizes de modelagem para os modelos funcionais e a atividade de inserção de marcas no modelo funcional podem ser consideradas limitações da AST, entretanto, esta questão pode ser facilmente contornada com um treinamento básico da equipe de desenvolvimento, apresentando para a mesma o processo de transformação de modelos proposto pela AST.

5.8 AVALIAÇÃO DA AST FRENTE AOS TRABALHOS RELACIONADOS

A AST explora a capacidade que as ferramentas de modelagem possuem de permitir o refinamento e a organização do projeto detalhado de um sistema de controle por meio blocos que armazenem internamente uma hierarquia de diagramas de blocos. Estes blocos permitem a identificação de aspectos arquiteturais de um sistema de controle, viabilizando assim a geração de componentes de software no modelo de arquitetura do sistema. É possível afirmar que a AST estende o que foi proposto por (RAGHAV et al., 2009b), uma vez que ela oferece suporte para um mapeamento entre diferentes combinações de ferramentas de modelagem funcional e de linguagens de descrição de arquitetura, e que além disso, prevê o mapeamento de aspectos estruturais, operacionais e comportamentais de um modelo funcional para um modelo de arquitetura do sistema. Na AST estes tipos de mapeamento estão diretamente relacionados com a posição hierárquica do diagrama de blocos dentro do modelo funcional e com a composição interna do mesmo.

O *plugin importer.simulink* e o mecanismo de transformação de modelos apresentado neste trabalho de pesquisa parecem ter o mesmo objetivo, apesar de aparentemente adotarem uma abordagem diferente para realizar os mapeamentos estrutural e comportamental entre os modelos Simulink e AADL. Um comparativo entre o *plugin importer.simulink* e a AST apontam que: (i) o projeto do *plugin importer.simulink* está em um estágio inicial de desenvolvimento se comparado com o projeto do mecanismo de transformação de modelos apresentado neste documento de tese, (ii) ele não suporta a importação de modelos Simulink em formato mdl, (iii) o mapeamento estrutural

gera apenas componentes do tipo *System* e (iv) o mapeamento comportamental suporta o mapeamento de blocos *Stateflow* hierárquicos. No entanto, o foco do mapeamento comportamental parece ser a identificação de problemas arquitetônicos relacionados ao compartilhamento de dados, e não a verificação de propriedades comportamentais como é o caso do mecanismo proposto por esta tese de doutorado. Além disso, parece que o *plugin importer.simulink* não suporta o mapeamento de possíveis modos de operação do sistema especificados em modelos funcionais Simulink para o modelo de arquitetura AADL correspondente.

A AST pode não ser uma solução definitiva para a questão da transformação de modelos funcionais Simulink em modelos de Arquitetura AADL, mas como pode ser observado na Tabela 7, ela oferece suporte a uma quantidade maior de recursos se comparada com os trabalhos relacionados apresentados no Capítulo 3 deste documento.

	ASSERT	**IME**	**Polychrony**	**Integração Códigos-Fonte**	**Plugin importer. simulink**	**AST**
Gera Modelo de Arquitetura a partir de Modelo Funcional?	não	sim	não	não	sim	sim
Prevê um mapeamento direto entre modelos Simulink e AADL?	não	não	não	não	sim	sim
Aplica a técnica de transformação de modelos M2M?	sim	sim	sim	não	sim	sim
Suporta mapeamento estrutural?	sim	sim	não	não	sim	sim
Suporta mapeamento comportamental?	não	não	não	não	sim	sim
Suporta mapeamento de modos de operação?	não	não	não	não	não	sim
Possui suporte de ferramenta computacional?	sim	sim	sim	não	sim	sim

Tabela 7 – Quadro Comparativo dos Trabalhos Relacionados

5.9 CONSIDERAÇÕES

Atualmente, o metamodelo Simulink não é utilizado para validar o modelo funcional Simulink de entrada, ou seja, não existe um mecanismo implementado no motor de transformação responsável por esta validação. Parte-se do princípio de que as diretrizes de modelagem foram seguidas, e que, portanto, o modelo de entrada é válido. Em relação ao metamodelo AADL, consideramos que o modelo AADL

gerado está consistente com o metamodelo AADL se o mesmo for reconhecido pelo editor de modelos AADL chamado OSATE.

É fato que o modelo AADL gerado AST é considerado preliminar, ou seja, o projetista precisa, posteriormente, especificar uma plataforma de execução que suporte a execução dos componentes da aplicação, e, também, inserir informações adicionais nos componentes do modelo AADL para que ele possa ser submetido a análises e verificações e consequentemente validado.

Embora neste trabalho de pesquisa tenha sido explorada de forma prática a transformação de modelos funcionais Simulink em modelos de arquitetura de software AADL, é possível inferir que a AST pode ser generalizada. Ou seja, a solução proposta também pode ser aplicada com outras de ferramentas modelagem funcional, desde que estas sejam baseadas em diagramas de blocos, como por exemplo, as ferramentas LabView e Scilab.

6 VALIDAÇÃO EXPERIMENTAL DA AST

O estudo de caso apresentado neste capítulo foi desenvolvido com o objetivo de avaliar o processo de transformação de modelos proposto pela AST. Para tal propósito, foi utilizado como entrada o modelo funcional Simulink de um Veículo Aéreo Não Tripulado (VANT).

O modelo funcional do VANT foi desenvolvido por integrantes do projeto ProVant[1]. O projeto ProVant está em desenvolvimento no Departamento de Automação e Sistemas (DAS) da Universidade Federal de Santa Catarina (UFSC), e tem como objetivo o desenvolvimento de uma aeronave não tripulada de pequena escala capaz de realizar voos de maneira autônoma (vide (GONCALVES et al., 2013)).

6.1 VISÃO GERAL DO PROJETO PROVANT

O VANT em desenvolvimento deve ser capaz de realizar voos autônomos de acordo com missões pré-definidas, ou seja, o VANT deve ser capaz de receber como entrada uma determinada trajetória e a seguir automaticamente. A missão é definida pelo operador do sistema através de uma aplicação específica, que é executada em um computador remoto denominado estação base (BS). A BS se comunica com o VANT por meio de uma interface de comunicação sem fio. Além de permitir a definição da missão, através da BS também é possível:

- Iniciar um voo autônomo para executar uma missão.
- Abortar uma missão por meio de duas ações distintas: *(i) retornar para base* ou *(ii) pouso de emergência*.
- Monitorar informações do voo (telemetria).
- Realizar testes individuais de dispositivos eletro/eletrônicos.

Cada uma destas ações deve desencadear uma operação diferente no VANT. Estas operações são executadas na plataforma de computação embarcada que compõe o VANT. Antes de iniciar um voo autônomo é necessário carregar uma missão para o VANT e confirmar o início da missão. Depois disso é necessário executar alguns subsistemas, tais como, controle de estabilidade, controle de trajetória, monitoramento da bateria e procedimentos de pouso e decolagem.

[1] http://provant.das.ufsc.br

Abortar uma missão pode ser visto como uma ação extraordinária, excepcional, que pode acontecer devido a várias razões, que vão desde más condições meteorológicas até uma descoberta de falha no sistema. Duas ações diferentes podem ser acionadas no caso de abortar uma missão, sendo que a primeira é identificada como *retornar para base*, onde o VANT deve retornar a uma localização geográfica específica (normalmente perto da BS). A outra ação é um possível *pouso de emergência* o mais próximo possível da localização corrente do VANT.

O monitoramento de informações do voo (telemetria) é uma ação opcional que pode ou não ser solicitada pelo operador. Através dessa opção, o operador pode manter uma vigilância constante no VANT e tomar decisões imediatas em caso de eventos extraordinários.

Finalizando, os testes individuais de dispositivos eletro/eletrônicos do VANT podem ser realizados como uma ação preventiva, semelhante ao que os pilotos de avião fazem antes de iniciar seus voos. Esta ação desencadeia uma rotina de testes no VANT para assegurar a integridade do sistema, bem como a verificação do canal de comunicação com a estação base.

Em relação a infraestrutura computacional do VANT, a mesma é composta por um computador pessoal (estação base) e pela plataforma embarcada do VANT, a qual consiste da placa denominada *Beaglebone*, equipada com um microcontrolador ARM Cortex-A8 superescalar de 720MHz, e de uma placa *Discovery*, equipada com um microcontrolador ARM Cortex-M4F de 168MHz. O transmissor *MRF24J40MC* da *Microchip* é utilizado para suportar a comunicação do VANT com a BS. Tal transmissor usa o padrão *IEEE. 802.15.4 2.4 GHz* e possui um modo de transmissão avançado capaz de atingir uma distância de até 1.200 metros. Em relação aos periféricos, são utilizados os seguintes sensores e atuadores:

- 1 Sonar
- 1 GPS
- 1 Inertial Measurement Unit (IMU)
- 2 Brushless-controllers (ESCs)
- 2 Servo-motors

O projeto PROVANT vem sendo conduzido levando em consideração o processo de desenvolvimento descrito na Seção 4 deste

documento, cuja representação gráfica é apresentada novamente na Figura 47.

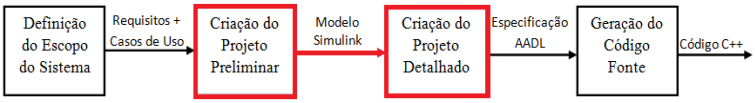

Figura 47 – Principais atividades e artefatos do método para desenvolver CPSs.

6.2 APLICAÇÃO DA AST NO PROJETO PROVANT

A aplicação da AST durante o processo de desenvolvimento do VANT ocorreu durante a transição entre a etapa denominada *Projeto Preliminar* e a etapa do *Projeto Detalhado*. Mais especificamente, a aplicação da AST deu suporte para fazer a transição entre modelo funcional Simulink e o modelo de arquitetura de software em AADL do sistema. O modelo funcional Simulink do VANT expressa a hierarquia estrutural dos subsistemas que compõem o sistema de controle, suas interfaces e conexões. O modelo funcional Simulink completo do VANT é resultado de um trabalho conjunto entre um engenheiro de controle e um engenheiro de computação, membros integrantes do projeto *ProVANT*, e entre outras finalidades, é utilizado para testar e simular os algoritmos de controle desenvolvidos ao longo do projeto. Conforme discutido ao longo deste capítulo, a aplicação da AST facilitou consideravelmente a geração de uma versão preliminar do modelo AADL do sistema do VANT a partir de informações extraídas do modelo funcional Simulink.

Detalhando a aplicação da AST, o processo teve início com o mapeamento do primeiro nível hierárquico do modelo funcional Simulink, o qual é ilustrado na Figura 48. Neste nível, o modelo funcional Simulink é composto por três blocos *Subsystem*, sendo um do tipo *Simple* e dois do tipo *Composite*. Um dos blocos *Subsystem* representa o sistema de controle do UAV propriamente dito, um representa a aplicação da Estação Base, e o outro representa um dispositivo de Controle Remoto que pode enviar comandos diretamente para o VANT. Para permitir a aplicação da AST, todos os blocos *Subsystem* que compõem o modelo funcional Simulink do VANT foram devidamente marcados. É possível conferir as marcações efetuadas, tanto no modelo da Figura 48 quanto nos próximos, observando a parte inferior dos blocos *Subsystem*.

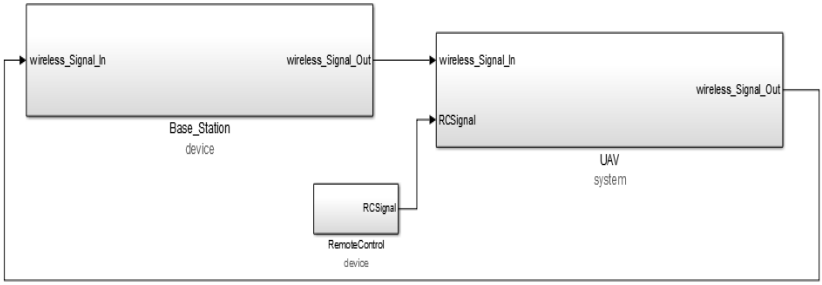

Figura 48 – Modelo Funcional Simulink - VANT

A Figura 49 mostra o segundo nível hierárquico do modelo funcional Simulink, correspondente ao refinamento do bloco *UAV* do modelo nível 1 apresentado na figura 48. Neste nível, o modelo possui 12 blocos *Subsystem*, sendo um do tipo *Stateflow* identificado como *OperationModes* e 10 blocos do tipo *Composite*. O bloco *SensorInterface* é responsável por receber os dados dos diversos sensores e encaminhá-los para o bloco *DataProcessing*, que tem como responsabilidade fusionar os dados e gerar as informações necessárias ao bloco *StabilizationControl*. O bloco *StabilizationControl* é responsável por estabilizar a aeronave. O bloco *TrajectoryControl* é responsável por fazer o UAV percorrer os *waypoints* previamente definidos pela trajetória especificada pela missão. Já o bloco *ControlMission* recebe a missão definida pelo usuário via estação base e emite a trajetória que deve ser percorrida pelo UAV. O bloco *RCSignalDecoder* recebe os sinais emitidos pelo controle remoto e repassa para o bloco *RCControl*, que estará ativo quando o VANT estiver operando em modo manual. O bloco *WirelessModule* é responsável por gerenciar a comunicação entre o VANT e a BS. Como a plataforma de execução do VANT prevê a utilização de dois processadores, os blocos *BeagleCommunication* e *DiscoreveryCommuncation* são responsáveis pela comunicação entre os blocos alocados em cada um destes processadores. O bloco subsystem *UAV_Dinamycs* representa a planta do VANT, ou seja, as equações que descrevem o comportamento da estrutura eletromecânica que constitui a aeronave.

O diagrama *Stateflow* do bloco *OperationModes* (Figura 49) é apresentado na figura 50. Ele é formado por seis estados básicos denominados *Neutral, Automatic_Mode, RadioControled_Mode, Maintenance_Mode, EmergencyLanding_Mode, ReturnToHome_Mode*. Estes estados representam os modos de operação do sistema e possuem um

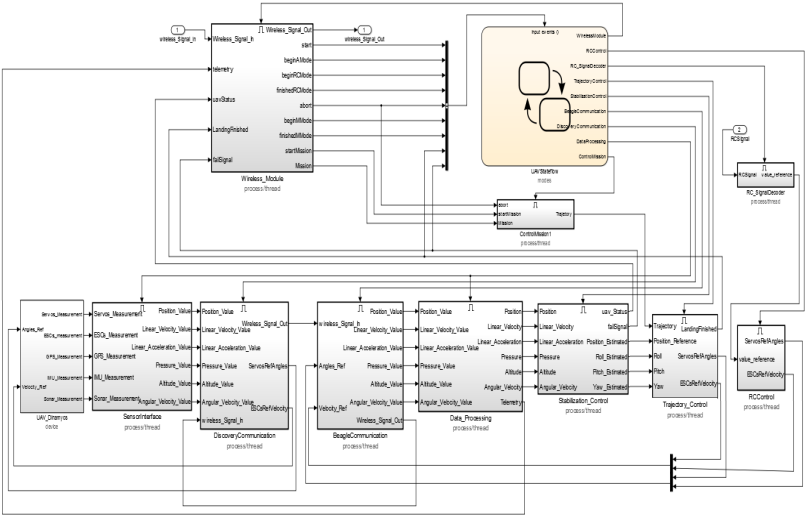

Figura 49 – Modelo Funcional Simulink de Nível 2 - Refinamento do bloco *VANT*.

conjunto de transições ativadas exclusivamente por eventos. Os estados possuem ações que ativam determinados blocos em cada modo de operação do sistema.

A Figura 51 apresenta o diagrama *Stateflow* que está posicionado hierarquicamente dentro do bloco subsystem *ControlMission*, estando, portanto, no terceiro nível hierárquico do modelo. Este diagrama tem como finalidade especificar o comportamento esperado da funcionalidade do sistema responsável por controlar a execução da missão recebida pelo UAV.

Para contribuir com a compreensão das vantagens da aplicação da AST durante o processo de desenvolvimento de um CPS de modo geral, vale destacar que o modelo de arquitetura do sistema em AADL do VANT apresentado a partir deste ponto do documento foi gerado a partir da extração de informações estruturais, operacionais (modos de operação) e comportamentais do modelo funcional Simulink apresentado. O modelo AADL do VANT foi gerado de forma automática, utilizando o protótipo da ferramenta que automatiza o uso da AST.

O modelo AADL de nível 1 mostrado na Figura 52 foi gerado pelo mapeamento estrutural da AST aplicado sobre o modelo funcional Simulink de nível 1 apresentado na Figura 48. É possível observar

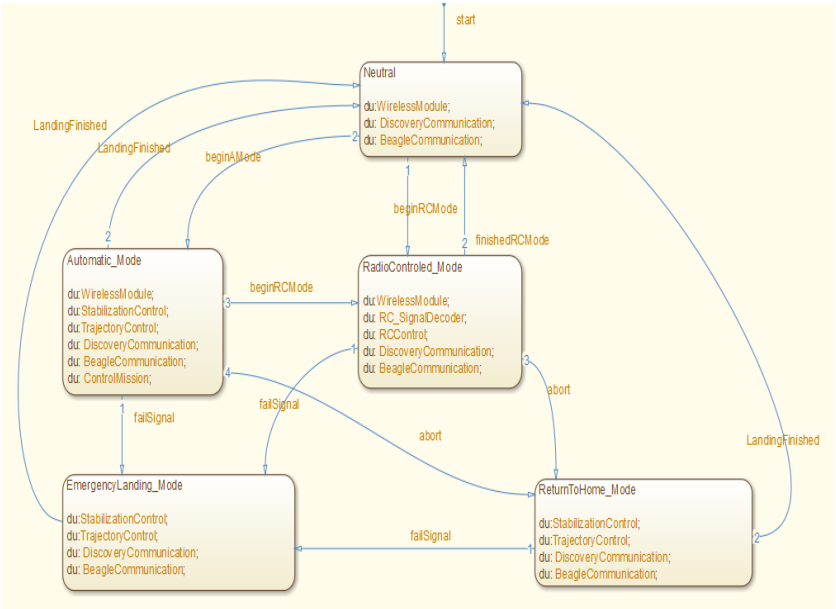

Figura 50 – Diagrama Stateflow do bloco Stateflow ModosDeOperação

que o modelo AADL gerado contém um componente *System* raiz chamado *arquitetura_uav.impl* que possui como subcomponentes dois *devices* chamados de *d_remotecontrol* e *d_basestation* (linhas 3 e 4) e um outro componente system chamado de *s_uavsystem* (linha 5). Além dos subcomponentes, o sistema raiz possui as conexões mapeadas a partir das linhas do modelo funcional Simulink de nível 1 (linhas 6 a 9). O device *d_remotecontrol*, o device *d_basestation* e o sistema *s_uavsystem* foram gerados pelo mapeamento estrutural a partir das marcas *device* e *system* inseridas nos blocos *Subsystem*. As conexões foram geradas a partir das linhas existentes entre os blocos *Subsystem* do modelo funcional Simulink do VANT.

O modelo AADL de nível 2 gerado pelo mapeamento estrutural da AST é mostrado na Figura 53. De acordo com as marcas recebidas, todos os blocos *Subsystem* do modelo funcional Simulink de Nível 2 (Figura 49) foram mapeados estruturalmente no modelo AADL como subcomponentes do componente *System* denominado *s_uav.impl* (linhas 2 a 26). As linhas entre os blocos *Subsystem* do modelo funcional de Nível 2 (Figura 49) foram mapeadas estruturalmente como conexões no modelo AADL (linhas 28 a 37).

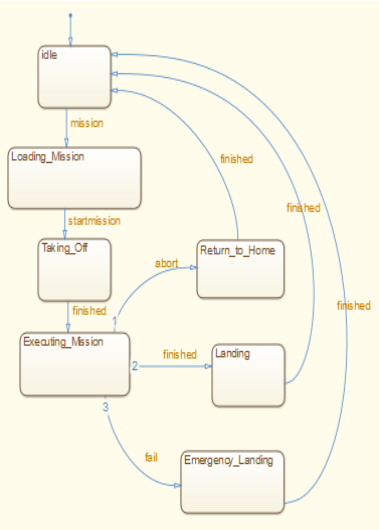

Figura 51 – Diagrama Stateflow do bloco ControlMission

```
1 SYSTEM IMPLEMENTATION uav.impl
2 SUBCOMPONENTS
3   d_base_station: DEVICE d_base_station.impl;
4   d_remotecontrol: DEVICE d_remotecontrol.impl;
5   s_uav: SYSTEM s_uav.impl;
6 CONNECTIONS
7     C1: PORT d_base_station.wireless_signal_out -> s_uav.wireless_signal_in;
8     C2: PORT s_uav.wireless_signal_out -> d_base_station.wireless_signal_in;
9     C3: PORT d_remotecontrol.rcsignal -> s_uav.rcsignal;
10 END uav.impl;
```

Figura 52 – Modelo AADL de Nível 1 - VANT

O mapeamento dos modos de operação da AST, baseado no diagrama *Stateflow* do bloco *UAVStateflow* do modelo funcional Simulink de nível 2, gerou a máquina de estados modal do componente *System s_uav.impl* no modelo AADL, conforme exibido nas linhas 39 a 45 da Figura 53. As transições de estado do diagrama *Stateflow* geraram as transições entre os modos de operação da máquina de estados modal, conforme exibido nas linhas 47 a 54 da Figura 53. A exemplo do que acontece no diagrama *Stateflow* da figura 50, os processos são ativados de acordo com o modo de operação ativo do sistema. Em AADL, o comando *in modes* é utilizado para especificar em que modo de operação cada processo do componente *system s_uavsystem* deve ser ativado.

Neste estudo de caso, o mapeamento estrutural da AST também gerou um subcomponente *thread* para cada um dos processos.

```
1 SYSTEM IMPLEMENTATION s_uav.impl
2  SUBCOMPONENTS
3    p_beaglecommunication: PROCESS p_beaglecommunication.impl in modes
4      (m_neutral, m_automatic_mode, m_radiocontroled_mode, m_emergencylanding_mode,
5       m_returntohome_mode);
6    p_controlmission: PROCESS p_controlmission.impl in modes
7      (m_automatic_mode, m_emergencylanding_mode, m_returntohome_mode);
8    p_data_processing: PROCESS p_data_processing.impl in modes
9      (m_neutral, m_automatic_mode, m_radiocontroled_mode, m_emergencylanding_mode,
10      m_returntohome_mode);
11   p_discoverycommunication: PROCESS p_discoverycommunication.impl in modes
12     (m_neutral, m_automatic_mode, m_radiocontroled_mode, m_emergencylanding_mode,
13      m_returntohome_mode);
14   p_rccontrol: PROCESS p_rccontrol.impl in modes(m_radiocontroled_mode,
15     m_emergencylanding_mode, m_returntohome_mode);
16   p_rc_signaldecoder: PROCESS p_rc_signaldecoder.impl in modes(m_radiocontroled_mode,
17      m_emergencylanding_mode, m_returntohome_mode);
18   p_sensorinterface: PROCESS p_sensorinterface.impl in modes (m_neutral, m_automatic_mode,
19      m_radiocontroled_mode, m_emergencylanding_mode, m_returntohome_mode);
20   p_stabilization_control: PROCESS p_stabilization_control.impl in modes(m_automatic_mode,
21     m_emergencylanding_mode, m_returntohome_mode);
22   p_trajectory_control: PROCESS p_trajectory_control.impl in modes(m_automatic_mode,
23     m_emergencylanding_mode, m_returntohome_mode);
24   d_uav_dinamycs: DEVICE d_uav_dinamycs.impl;
25   p_wireless_module: PROCESS p_wireless_module.impl in modes(m_neutral, m_automatic_mode,
26     m_radiocontroled_mode, m_emergencylanding_mode, m_returntohome_mode);
27
28  CONNECTIONS
29    C1: PORT p_wireless_module.wireless_signal_out -> wireless_signal_out;
30    C2: PORT wireless_signal_in -> p_wireless_module.wireless_signal_in;
31    C3: PORT d_uav_dinamycs.servoangle -> p_data_processing.servoangle;
32    C4: PORT d_uav_dinamycs.rotorspeed -> p_data_processing.rotorspeed;
33    C5: PORT d_uav_dinamycs.heading -> p_data_processing.heading;
34    C6: PORT d_uav_dinamycs.altitude -> p_data_processing.altitude;
35    C7: PORT d_uav_dinamycs.yaw -> p_data_processing.yaw;
36    C8: PORT p_stabilization_control.yaw -> p_trajectory_control.yaw;
37    ... o restante das conexões foi suprimido
38
39    MODES
40    m_returntohome_mode: mode;
41    m_maintenance_mode: mode;
42    m_raciocontroled_mode: mode;
43    m_emergencylanding_mode: mode;
44    m_automatic_mode: mode;
45    m_neutral: initial mode;
46
47    m_neutral -[p_wireless_module.start]-> m_neutral;
48    m_returntohome_mode -[p_wireless_module.landingfinished]-> m_neutral;
49    m_returntohome_mode -[p_trajectory_control.landingfinished]-> m_neutral;
50    m_emergencylanding_mode -[p_wireless_module.landingfinished]-> m_neutral;
51    m_emergencylanding_mode -[p_trajectory_control.landingfinished]-> m_neutral;
52    m_automatic_mode -[p_wireless_module.landingfinished]-> m_neutral;
53    m_automatic_mode -[p_trajectory_control.landingfinished]-> m_neutral;
54    ... o restante das transições foi suprimido
55 END s_uav.impl;
```

Figura 53 – Modelo AADL de Nível 2 - VANT

```
1  PROCESS p_stabilization_control
2    FEATURES
3      uavstatus: OUT DATA PORT Base_Types_Simulink::integer;
4      failsignal: OUT DATA PORT Base_Types_Simulink::integer;
5      velocity_reference: OUT DATA PORT Base_Types_Simulink::integer;
6      position_reference: OUT DATA PORT Base_Types_Simulink::integer;
7      roll: OUT DATA PORT Base_Types_Simulink::integer;
8      pitch: OUT DATA PORT Base_Types_Simulink::integer;
9      yaw: OUT DATA PORT Base_Types_Simulink::integer;
10     position_estimated: IN DATA PORT Base_Types_Simulink::integer;
11     linearvelocity: IN DATA PORT Base_Types_Simulink::integer;
12     altitude_estimated: IN DATA PORT Base_Types_Simulink::integer;
13     angularvelocity: IN DATA PORT Base_Types_Simulink::integer;
14 END p_stabilization_control;
15
16 PROCESS IMPLEMENTATION p_stabilization_control.impl
17   SUBCOMPONENTS
18      t_stabilization_control: THREAD t_stabilization_control.impl;
19   CONNECTIONS
21      C1: PORT position_estimated -> t_stabilization_control.position_estimated;
22      C2: PORT linearvelocity -> t_stabilization_control.linearvelocity;
23      C3: PORT altitude_estimated -> t_stabilization_control.altitude_estimated;
24      C4: PORT angularvelocity -> t_stabilization_control.angularvelocity;
25      C5: PORT t_stabilization_control.uavstatus -> uavstatus;
26      C6: PORT t_stabilization_control.failsignal -> failsignal;
27      C7: PORT t_stabilization_control.velocity_reference -> velocity_reference;
28      C8: PORT t_stabilization_control.position_reference -> position_reference;
29      C9: PORT t_stabilization_control.roll -> roll;
30      C10: PORT t_stabilization_control.pitch -> pitch;
31      C11: PORT t_stabilization_control.yaw -> yaw;
32 END p_stabilization_control.impl;
33 THREAD t_stabilization_control
34   FEATURES
35      uavstatus: OUT DATA PORT Base_Types_Simulink::integer;
36      failsignal: OUT DATA PORT Base_Types_Simulink::integer;
37      velocity_reference: OUT DATA PORT Base_Types_Simulink::integer;
38      position_reference: OUT DATA PORT Base_Types_Simulink::integer;
39      roll: OUT DATA PORT Base_Types_Simulink::integer;
40      pitch: OUT DATA PORT Base_Types_Simulink::integer;
41      yaw: OUT DATA PORT Base_Types_Simulink::integer;
42      position_estimated: IN DATA PORT Base_Types_Simulink::integer;
43      linearvelocity: IN DATA PORT Base_Types_Simulink::integer;
44      altitude_estimated: IN DATA PORT Base_Types_Simulink::integer;
45      angularvelocity: IN DATA PORT Base_Types_Simulink::integer;
46 END t_stabilization_control;
47 THREAD IMPLEMENTATION t_stabilization_control.impl
48   calls
49      Mycalls: {
50      P_Spg : subprogram programs_simulink::stabilization_control;
51      };
52 END t_stabilization_control.impl;
```

Figura 54 – Modelo AADL de Nível 3 - VANT

Com a finalidade de ilustrar a estrutura de um processo, a Figura 54 apresenta o modelo AADL de nível 3 gerado para o processo *p_Stabilization_Control*, onde o subcomponente *t_stabilization_control* pode ser visto na linha 18. Também é possível observar nesta figura que todas as portas do processo *p_Stabilization_Control* possuem os tipos de dados devidamente especificados (veja linhas 3 a 13). Além disso, existe a chamada a um *subprogram* feita pela thread *t_stabilization_control.impl*, conforme pode ser observado nas linhas 48 a 51.

O mapeamento comportamental realizado se baseou no diagrama *Stateflow* do bloco *ControlMission* do modelo de nível 2. O mesmo gerou a seção *annex behavior_specification* da thread *ControlMission*

no modelo AADL, conforme exibido nas as linhas 2 a 21 da Figura 55. Vale ressaltar que quando o mapeamento comportamental é executado, a seção *annex behavior_specification* é consequentemente gerada, sendo que o mapeamento estrutural não gera a chamada de um componente *subprogram* para esta thread. Isto acontece por que os dois recursos são mutuamente exclusivos, uma vez que ambos são utilizados em AADL para especificar o comportamento de uma thread.

```
1 THREAD IMPLEMENTATION t_control_mission.impl
2 annex behavior_specification{**
3 states
4   s_idle: initial complete state;
5   s_loading_mission: complete state;
6   s_taking_off: complete state;
7   s_execution_mission:complete state;
8   s_return_to_home: complete state;
9   s_landing:  complete state;
10  s_emergency_landing: complete state;
11 transitions
12    s_idle -[mission]-> s_loading_mission;
13    s_loading_mission -[startmission]-> s_taking_off;
14    s_taking_off[on dispatch -[finished]-> s_execution_mission;
15    s_execution_mission -[finished ]-> s_landing;
16    s_execution_mission -[abort]-> s_return_to_home;
17    s_execution_mission -[fail]-> s_emergency_landing;
18    s_landing -[finished]-> s_idle;        ]
19    s_return_to_home -[finished]-> s_idle;
20    s_emergency_landing -[finished]-> s_idle;
21 **};
22 END t_control_mission.impl;
```

Figura 55 – Modelo AADL de Nível 3 - thread *ControlMission*

6.3 VALIDAÇÃO DO MODELO AADL DO VANT

A validação de um modelo de arquitetura AADL pode ser feita de diversas formas. Validar a sintaxe é necessário, mas, além disso, é necessário validar outros aspectos do sistema, como, por exemplo, segurança, latência, escalonabilidade, etc. Entretanto, estas análises requerem que o modelo AADL possua uma plataforma de execução definida (processadores, memória, etc.). As informações referentes a plataforma de execução foram adicionadas manualmente no modelo AADL do VANT, uma vez as mesmas não são passíveis de geração automática a partir do modelo funcional.

A plataforma de execução do sistema do VANT, definida para a validação do modelo AADL gerado pela AST, é composta por 2 processadores e uma memória associada a cada um deles. Assumiu-se que ambos os processadores utilizam o protocolo de escalonamento *Rate Monotonic* (RM).

Para o estudo de caso do VANT foram realizadas análises de

escalonabilidade, de tempo de resposta e também a verificação de algumas propriedades comportamentais do modelo. A realização destas análises e verificações possibilitaram ajustar propriedades de execução e o comportamento esperado de determinados componentes de software do sistema em desenvolvimento. Alguns dos resultados obtidos a partir da realização destas análises e verificações são apresentados a seguir.

6.3.1 Análise de Escalonabilidade

Para que fosse possível realizar a análise de escalonabilidade foi necessário definir algumas propriedades de execução das *threads* e também alocar estas *threads* nos processadores que compõem a plataforma de execução do sistema. As Tabelas 8 e 9 apresentam a alocação das *threads* nos processadores, bem como algumas propriedades de execução das *threads*.

Propriedades/ Thread	**Wireless Module**	**Data Processing**	**Stabilization Control**	**Trajectory Control**	**Beagle Communication**
Processador	cortex_a8	cortex_a8	cortex_a8	cortex_a8	cortex_a8
Peridiocidade	Periódica	Periódica	Periódica	Periódica	Periódica
Período	10ms	10ms	3ms	12ms	10ms
Deadline	10ms	10ms	3ms	12ms	10ms

Tabela 8 – Características Plataforma de Execução

Propriedades/ Thread	**Discovery Communication**	**RCSignal Decoder**	**RCControl**	**Sensor Interface**	**Control Mission**
Processador	cortex_m4f	cortex_m4f	cortex_m4f	cortex_m4f	cortex_a8
Peridiocidade	Periódica	Periódica	Periódica	Periódica	Periódica
Período	10ms	10ms	10ms	10ms	10ms
Deadline	10ms	10ms	10ms	10ms	10ms

Tabela 9 – Características Plataforma de Execução - continuação

A Figura 56 apresenta o modelo AADL do VANT com a plataforma de execução devidamente especificada. Observe que nas linhas de 11 a 15 são definidos os dois processadores citados anteriormente, bem como uma memória para cada um deles. As linhas de 26 a 37 e de 49 a 55 apresentam a distribuição das *threads* em relação aos processadores da plataforma de execução, e as linhas de 40 a 47 e 58 a 63 apresentam a associação entre os processos e as memórias de cada um dos processadores.

A análise de escalonabilidade de modelos AADL requer que pelo menos as propriedades de execução: *dispatch_protocol, period, e deadline* das threads estejam especificadas. Observe as linhas de 14 a

```
1 SYSTEM IMPLEMENTATION s_uav.impl
2  SUBCOMPONENTS
3    p_beaglecommunication: PROCESS p_beaglecommunication.impl in modes
4      (m_neutral, m_automatic_mode, m_radiocontroled_mode, m_emergencylanding_mode,
5       m_returntohome_mode);
9    ... o restante dos processos foi suprimido
11    CORTEX_A8: processor execution_plataform::CORTEX_A8.impl;
12    Stand_Memory: memory execution_plataform::ram.impl;
13
14    CORTEX_CORTEX_M4F: processor execution_plataform::CORTEX_M4F.impl;
15    Stand_Memory2: memory execution_plataform::ram.impl;
16  CONNECTIONS
17    C1: PORT p_wireless_module.wireless_signal_out -> wireless_signal_out;
18   ... o restante das conexões foi suprimido
19  MODES
20    m_neutral: initial mode;
21    m_automatic_mode: mode;
22    ......
23    m_neutral -[p_wireless_module.start]-> m_neutral;
24    m_returntohome_mode -[p_wireless_module.landingfinished]-> m_neutral;
25    ......
26   PROPERTIES
27    Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
28                                p_wireless_module.t_wireless_module;
29    Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
30                                p_data_processing.t_data_processing;
31    Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
32                                p_stabilization_control.t_stabilization_control;
33    Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
34                                p_trajectory_control.t_trajectory_control;
35    Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
36                                p_beaglecommunication.t_beaglecommunication;
37    Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
38                                p_controlmission.t_control_mission;
39
40    Actual_Memory_Binding => (reference(Stand_Memory)) applies to p_wireless_module;
41    Actual_Memory_Binding => (reference(Stand_Memory)) applies to p_data_processing;
42    Actual_Memory_Binding => (reference(Stand_Memory)) applies to
43                                                        p_stabilization_control;
44    Actual_Memory_Binding => (reference(Stand_Memory)) applies to p_trajectory_control;
45    Actual_Memory_Binding => (reference(Stand_Memory)) applies to
46                                                        p_beaglecommunication;
47    Actual_Memory_Binding => (reference(Stand_Memory)) applies to p_controlmission;
48
49    Actual_Processor_Binding => (reference(CORTEX_M4F)) applies to
50                                p_discoverycommunication.t_discoverycommunication;
51    Actual_Processor_Binding => (reference(CORTEX_M4F)) applies to
52                                  p_rc_signaldecoder.t_rc_signaldecoder;
53    Actual_Processor_Binding => (reference(CORTEX_M4F)) applies to
54                                  p_rccontrol.t_rccontrol;
55    Actual_Processor_Binding => (reference(CORTEX_M4F)) applies to
56                                  p_sensorinterface.t_sensorinterface;
57
58    Actual_Memory_Binding => (reference(Stand_Memory2)) applies to
59                                p_discoverycommunication;
60    Actual_Memory_Binding => (reference(Stand_Memory2)) applies to p_rc_signaldecoder;
61    Actual_Memory_Binding => (reference(Stand_Memory2)) applies to p_rccontrol;
62    Actual_Memory_Binding => (reference(Stand_Memory2)) applies to p_sensorinterface;
63 END s_uav.impl;
```

Figura 56 – Modelo AADL de Nível 2 - UAV com a plataforma de execução definida

```
1  THREAD t_stabilization_control
2     FEATURES
3        uavstatus: OUT DATA PORT Base_Types_Simulink::integer;
4        failsignal: OUT DATA PORT Base_Types_Simulink::integer;
5        velocity_reference: OUT DATA PORT Base_Types_Simulink::integer;
6        position_reference: OUT DATA PORT Base_Types_Simulink::integer;
7        roll: OUT DATA PORT Base_Types_Simulink::integer;
8        pitch: OUT DATA PORT Base_Types_Simulink::integer;
9        yaw: OUT DATA PORT Base_Types_Simulink::integer;
10       position_estimated: IN DATA PORT Base_Types_Simulink::integer;
11       linearvelocity: IN DATA PORT Base_Types_Simulink::integer;
12       altitude_estimated: IN DATA PORT Base_Types_Simulink::integer;
13       angularvelocity: IN DATA PORT Base_Types_Simulink::integer;
14    PROPERTIES
15       dispatch_protocol => periodic;
16       period => 3 ms;
17       deadline => 3 ms;
18 END t_stabilization_control;
```

Figura 57 – Especificação AADL da thread *t_stabilization_control*

17 da Figura 57, este trecho de código apresenta a especificação das propriedades de execução da *thread t_stabilization_control*.

A Figura 58 apresenta o resultado da análise de escalonabilidade do modelo AADL do VANT realizada no OSATE, o qual está ligado a várias ferramentas de análise. Observando a Figura 58 é possível constatar que a partir do conjunto de propriedades definido, o sistema do VANT se mostrou escalonável, e, além disso, é possível observar o percentual de utilização de cada processador em cada modo de operação do sistema.

6.3.2 Análises de Fluxos de Latência

O modelo AADL do VANT também foi submetido à análise de tempo de resposta. Ou seja, foi analisado o tempo necessário para um sinal leva para viajar a partir de uma unidade de interface, trafegar pelo sistema de controle e retornar uma resposta. Para realizar tal análise, foi necessário adicionar no modelo AADL especificações de fluxo nos componentes individuais do sistema (*processos e threads*), e, além disso, especificar um fluxo *end-to-end* no sistema raiz do modelo AADL. Neste caso, foi analisado o tempo que o sinal responsável por transmitir a posição do UAV (valor do GPS) demora para percorrer os processos *SensorInterface*, *DiscoveryCommunication*, *BeagleCommunication* e *DataProcessing*, até chegar ao processo (*Stabilization_Control*) que precisa deste valor para estabilizar o UAV.

Observe que na linha 11 da Figura 59 foi especificada a origem do fluxo que foi analisado, e na linha 29 o destino do fluxo. O caminho que o fluxo percorre dentro do sistema do UAV pode ser observado nas

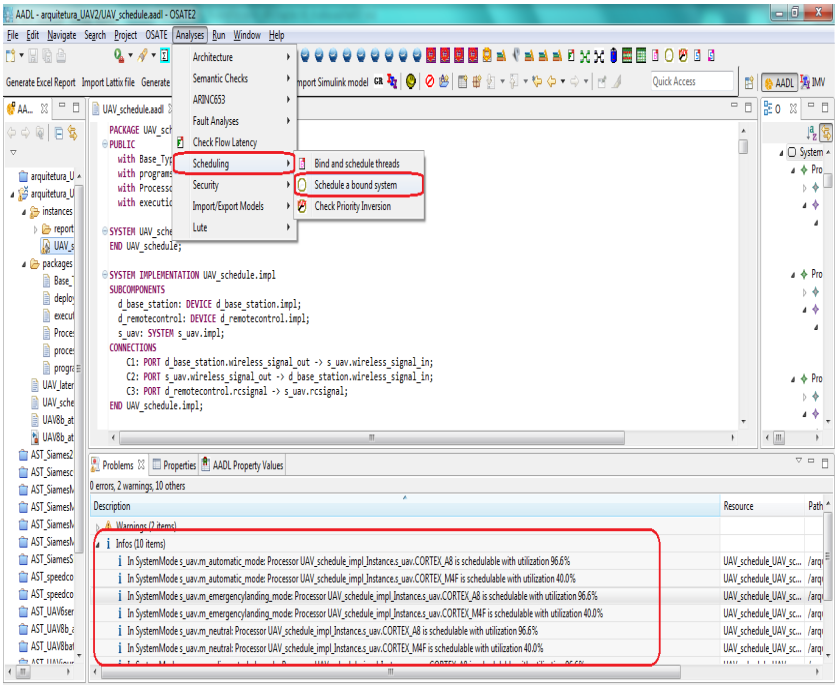

Figura 58 – Análise de Escalonabilidade do Modelo AADL do UAV

linhas 21 a 27 da Figura 60. Embora não tenha sido demonstrado, todo o caminho que o fluxo percorreu dentro dos processos *SensorInterface*, *DiscoveryCommunication*, *BeagleCommunication* e *DataProcessing* também precisou ser especificado, bem como a latência prevista para cada um dos trechos percorridos. A Figura 61 apresenta o resultado da análise de tempo de resposta especificada anteriormente, e a partir dela é possível observar que o tempo que o sinal leva para percorrer o caminho indicado não excede o limite de tempo especificado, que é de 70ms.

6.3.3 Verificação de Propriedades Comportamentais

É possível verificar a corretude do comportamento de um modelo AADL a partir da verificação formal de propriedades. Dentre os pontos passíveis de verificação, destacam-se as propriedades comportamentais

```
1 DEVICE d_uav_dinamycs
2  FEATURES
3    servos_measurement: OUT DATA PORT Base_Types_Simulink::integer;
4    escs_measurement: OUT DATA PORT Base_Types_Simulink::integer;
5    gps_measurement: OUT DATA PORT Base_Types_Simulink::integer;
6    imu_measurement: OUT DATA PORT Base_Types_Simulink::integer;
7    sonar_measurement: OUT DATA PORT Base_Types_Simulink::integer;
8    angles_ref: IN DATA PORT Base_Types_Simulink::integer;
9    velocity_ref: IN DATA PORT Base_Types_Simulink::integer;
10  flows
11    position_flow_src: flow source gps_measurement {latency => 5ms.. 5ms;};
12 END d_uav_dinamycs;
13
14 PROCESS p_stabilization_control
15  FEATURES
16    uav_status: OUT DATA PORT Base_Types_Simulink::integer;
17    failsignal: OUT DATA PORT Base_Types_Simulink::integer;
18    position_estimated: OUT DATA PORT Base_Types_Simulink::integer;
19    roll_estimated: OUT DATA PORT Base_Types_Simulink::integer;
20    pitch_estimated: OUT DATA PORT Base_Types_Simulink::integer;
21    yaw_estimated: OUT DATA PORT Base_Types_Simulink::integer;
22    position: IN DATA PORT Base_Types_Simulink::integer;
23    linear_velocity: IN DATA PORT Base_Types_Simulink::integer;
24    linear_acceleration: IN DATA PORT Base_Types_Simulink::integer;
25    pressure: IN DATA PORT Base_Types_Simulink::integer;
26    altitude: IN DATA PORT Base_Types_Simulink::integer;
27    angular_velocity: IN DATA PORT Base_Types_Simulink::integer;
28  flows
29    position_flow_snk: flow sink position {latency => 8 ms .. 8 ms;};
30 END p_stabilization_control;
```

Figura 59 – Especificação do caminho percorrido por um fluxo no modelo AADL

das threads, tais como vivacidade, ausência de bloqueio, justiça, etc. Entretanto, para isso, é preciso especificar o comportamento dos *devices* e das *threads* utilizando o anexo comportamental da AADL, e posteriormente utilizar a cadeia de transformação de modelos disponível no ambiente Topcased, a qual foi apresentada no Capítulo 2 deste documento.

A Tabela 10 apresenta o resultado de três diferentes verificações comportamentais executadas sobre o comportamento da thread *ControlMission* do modelo AADL. O comportamento desta thread foi especificado utilizando o anexo comportamental da AADL, gerado automaticamente a partir da execução do mapeamento comportalmental da AST. O modelo gerado pela AST foi submetido ao *plugin* AAADL2FIACRE da cadeia de verificação do ambiente Topcased, posteriormente as propriedades de verificação puderam ser descritas utilizando a lógica temporal LTL e processadas pelo *model checker* SELT. A primeira propriedade verifica a ausência de *deadlock* e a segunda verifica a ausência de divergência temporal. A terceria propriedade verifica se quando atingido, estado *Taking-Off* (decolando) bloqueia o comportamento da thread, neste caso em particular, a verificação retornou "*FALSE*" e apresentou um contra-exemplo, este resultado demonstra que o estado em questão não bloqueia o comportamento

```
1 SYSTEM IMPLEMENTATION s_uav.impl
2  SUBCOMPONENTS
3    p_beaglecommunication: PROCESS p_beaglecommunication.impl in modes
4      (m_neutral, m_automatic_mode, m_radiocontroled_mode, m_emergencylanding_mode,
5       m_returntohome_mode);
9    ... o restante dos processos foi suprimido
10
11   CORTEX_A8: processor execution_plataform::CORTEX_A8.impl;
12   Stand_Memory: memory execution_plataform::ram.impl;
13
14   CORTEX_M4F: processor execution_plataform::CORTEX_M4F.impl;
15   Stand_Memory2: memory execution_plataform::ram.impl;
16
17  CONNECTIONS
18   C1: PORT p_wireless_module.wireless_signal_out -> wireless_signal_out;
19   ... o restante das conexões foi suprimido
20
21  flows
22   position_end_to_end: end to end flow d_uav_dinamycs.position_flow_src -> c45 ->
23   p_sensorinterface.position_flow_path -> C32  ->
24     p_discoverycommunication.position_flow_path1 -> C29
25   -> p_beaglecommunication.position_flow_path2 -> c38 ->
26     p_data_processing.position_flow_path3 -> C4 ->
27   p_stabilization_control.position_flow_snk {latency => 70 ms .. 70 ms;};
28
29  MODES
30   m_neutral: initial mode;
31   m_automatic_mode: mode;
32   ... o restante das conexões foi suprimido
33   m_neutral -[p_wireless_module.start]-> m_neutral;
34   m_returntohome_mode -[p_wireless_module.landingfinished]-> m_neutral;
35   ... o restante das conexões foi suprimido
36
37 PROPERTIES
38   Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
39                               p_wireless_module.t_wireless_module;
40   Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
41                               p_data_processing.t_data_processing;
42   Actual_Processor_Binding => (reference(CORTEX_A8)) applies to
43                               p_stabilization_control.t_stabilization_control;
44   ... o restante foi suprimido
45 END s_uav.impl;
```

Figura 60 – Especificação do caminho percorrido por um fluxo no modelo AADL

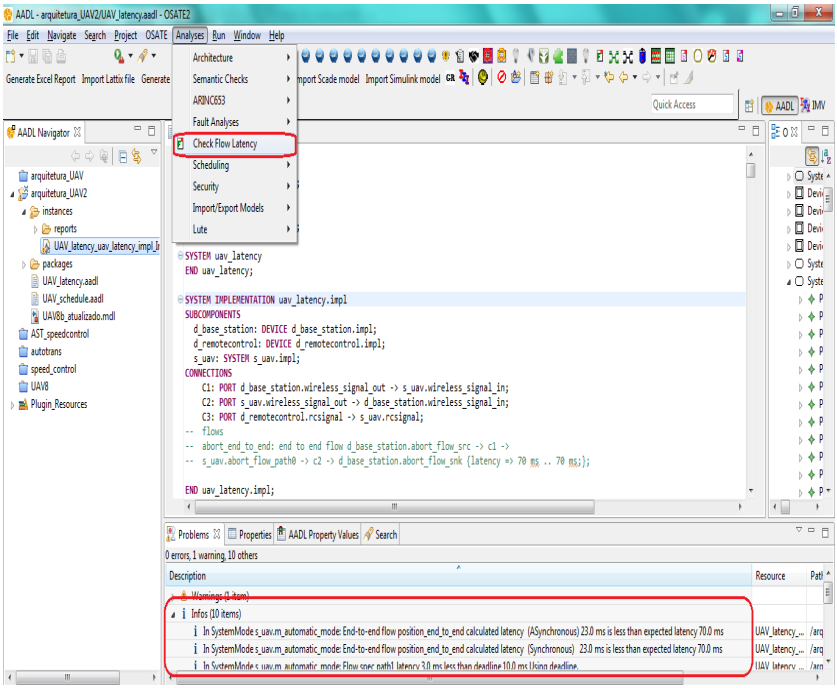

Figura 61 – Análise de Tempo de Resposta no Modelo AADL do UAV

da thread, e o *model-checker* SELT apresenta um caminho para provar tal afirmação.

Propriedade	Retorno SELT	Contra-Exemplo
-dead	TRUE	
-div	TRUE	
[]startmission	FALSE	STATE 0: -mission...(preserving T) ->STATE 4: -finished . . . (preserving - startmission) ->STATE5: [accepting all]

Tabela 10 – Resultados das Verificações das Propriedades Comportamentais

Além das análises realizadas até o momento, e conforme já discutido, outros tipos de análises e verificações estão disponíveis para modelos AADL. Dentre elas podemos citar a análise de falhas, a análise de carga da largura de banda para as conexões, a análise de carga total de banda nos barramentos, a análise de consumo de energia nos

barramentos, a análise de alocação de recursos para processadores e memórias, análise do orçamento total de recursos para os processadores e memórias, a verificação de incompatibilidade entre conexões, a verificação da existência de portas para conexões requeridas, a verificação de inversão de prioridade considerando à prioridade dos componentes atribuída pelo usuário, a verificação de *security* e de *safety* entre outras. Considerando que o projeto ProVant ainda está em desenvolvimento, é provável que o modelo de arquitetura do mesmo (modelo AADL) seja submetido a outros tipos de análises e verificações.

6.4 AVALIAÇÃO DA TRANSFORMAÇÃO DE MODELOS

Esta seção apresenta os resultados de uma avaliação do processo de transformação de modelos proposto pela AST, avaliação esta, que leva em consideração os critérios de correção, completude, singularidade, terminação, escalabilidade, usabilidade, reutilização, precisão, legibilidade, parcimônia, rastreabilidade, reversibilidade e nível de automação, apresentados anteriormente na seção 2.1.3 deste documento. Estes critérios foram avaliados durante o desenvolvimento dos estudos de caso do projeto SIAMES (apresentado no Capítulo 5) e do projeto ProVant (apresentado neste Capítulo).

- **Correção**. O processador de modelos AADL OSATE demonstrou que o modelo AADL gerado pela AST está em conformidade com a gramática da AADL, ou seja, ele está em conformidade com a especificação do metamodelo da AADL. Sendo assim, a transformação de modelos realizada pela AST é sintaticamente correta. No caso do mapeamento operacional e comportamental, a AST também preserva o comportamento especificado no modelo funcional no modelo de arquitetura. Nestes casos, a transformação de modelos da AST é semânticamente correta.

- **Completude**. Para cada elemento marcado no modelo funcional Simulink (bloco *Subsystem*) existe um elemento correspondente no modelo de arquitetura gerado pela AST. Sendo assim, a transformação de modelos da AST pode ser considerada completa.

- **Singularidade**. A transformação de modelos da AST fornece singularidade, isso porque ela gera um modelo de arquitetura AADL exclusivo para cada modelo funcional Simulink importado. Além disso, de acordo com a marcação feita no modelo funcional

Simulink importado, a AST gera diferentes modelos de arquitetura AADL.

- **Terminação**. O modelo de arquitetura vai ser gerado pela AST independentemente da aplicação das diretrizes de modelagem no modelo funcional importado, entretanto, neste caso, o modelo de arquitetura gerado pela AST pode requerer uma quantidade maior ajustes adicionais. Sendo assim, a transformação de modelos da AST fornece terminação, uma vez que ela sempre termina e leva a um resultado.

- **Legibilidade**. As regras de transformação apresentadas no Capítulo 5 são simples, legíveis e compreensíveis, prova disto, é que elas puderam ser automatizadas no plugin AS2T. Sendo assim, a transformação de modelos sugerida fornece legibilidade.

- **Escalabilidade**. Considerando o modelo funcional Simulink do SIAMES apresentado no capítulo 5 e modelo funcional Simulink do VANT apresentado neste capítulo, além de outros modelos funcionais Simulink que foram testados, a AST mostrou-se capaz de operar modelos simples e mais complexos sem sacrificar o desempenho. Portanto, é possível afirmar que transformação de modelos proposta pela AST possui escalabilidade.

- **Usabilidade**. O desenvolvimento dos estudos de caso demonstrou que a transformação de modelos proposta pela AST pode ser facilmente executada, tanto de forma manual quanto de forma automática, desde que o modelo funcional Simulink esteja devidamente estruturado e marcado. Isso demonstrou que a transformação de modelos sugerida é utilizável.

- **Precisão**. A transformação de modelos proposta pela AST é considerada precisa, uma vez que a AST possui a capacidade de importar modelos funcionais incompletos. Embora a AST suporte importar modelos funcionais Simulink incompletos, é recomendado que o modelo importado pela AST seja um modelo completo, isso porque o modelo AADL gerado pela AST reflete a estrutura hierárquica do modelo funcional Simulink importado, estando ele completo ou incompleto.

- **Parcimônia**. Como o modelo AADL gerado pela AST é um modelo simples e sem redundância, a transformação de modelos sugerida pela AST possui um bom nível de parcimônia.

- **Rastreabilidade**. A AST não oferece, até este momento, a capacidade de rastrear as ligações entre os elementos do modelo do modelo funcional Simulink importado e o modelo de arquitetura AADL gerado.

- **Reversibilidade**. Até este ponto da pesquisa, as regras de transformação da AST são unidirecionais, ou seja, elas especificam como ocorre a transformação de um componente do modelo funcional Simulink em um componente de software do modelo de arquitetura em AADL.

- **Nível de Automação**. O nível de automação da AST é alto, uma vez que foi possível automatizar todas as regras de transformação definidas no Capítulo 5. As informações adicionais que precisam ser inseridas no modelo de arquitetura AADL gerado pela AST são informações que não se encontram no modelo funcional Simulink, portanto, não é possível mapeá-las.

6.5 CONSIDERAÇÕES

O desevolvimento do estudo de caso apresentado neste Capítulo demonstrou que usando a AST é possível criar um modelo AADL que especifica a arquitetura de software de um CPS, a partir de informações extraídas de um modelo funcional Simulink previamente especificado. Tanto no caso do projeto ProVant quanto em testes complementares, as *"correlações âncora"* foram devidamente identificadas pelo mecanismo de transformação de modelos da AST. Os poucos casos em que isso não foi possível, vêm de modelos Simulink planos.

Grande parte do trabalho e do tempo empregados para especificar os componentes de software do modelo de arquitetura do sistema é reduzido, uma vez que o modelo AADL preliminar pode ser gerado de forma automática utilizando o *plugin* AS2T. No caso do projeto ProVant, o *plugin* AS2T gerou um modelo AADL modularizado, com 669 linhas de código, 1 sistema raiz, 3 *devices*, 1 subsistema, 10 processos, 17 *threads*, 17 subprogramas, 195 portas e 189 conexões. O modelo AADL do Vant após a inserção da plataforma de execução e das propriedades de execução dos componentes de software ficou com 810 linhas de código, sendo assim, a utilização do *plugin* AS2T gerou de forma automática cerca de 80% do modelo de arquitetura do sistema.

A AST tende a diminuir, principalmente se for aplicada de forma automatizada, a incidência de inconsistências na definição das

conexões entre as portas dos componentes de software, além de garantir a consistência dos dados entre as portas envolvidas nestas conexões. Além disso, possibilita a integração e a consistência entre os modelos funcional Simulink e de arquitetura AADL. Ou seja, a AST busca garantir a consistência entre o modelo AADL e os todos modelos Simulink (blocos *Subsystem*) associados aos componentes do modelo AADL.

Considerando que um modelo de arquitetura AADL pode ser submetido a vários tipos de análises e verificações não disponíveis para modelos projetados com a ferramenta Simulink, a AST contribui com a integração de aplicações verificadas em arquiteturas validadas, visando tornar o processo de desenvolvimento de CPSs mais robusto e confiável.

7 CONCLUSÕES E PERSPECTIVAS

Esse trabalho de tese propôs uma forma de abordar o projeto do sistema computacional embarcado de um CPS baseada na modelagem funcional do mesmo. A abordagem proposta, a AST, fundamenta-se em técnicas da MDE para sugerir como relacionar os elementos de um modelo funcional com os elementos de um modelo de arquitetura do sistema.

Considerando que durante a concepção de um CPS é comum a utilização de linguagens de modelagem de alto nível, como Simulink, para a modelagem dos requisitos funcionais do sistema, e a utilização de uma linguagem em nível de arquitetura, como a AADL, para modelar os aspectos não funcionais do mesmo sistema. A solução apresentada nesta tese, a AST, viabiliza a transformação de modelos Simulink, desenvolvidos para fins de simulação das funcionalidades esperadas do sistema, em modelos de arquitetura de software expressos em AADL, contribuindo assim com a composição do projeto do sistema computacional embarcado do CPS em desenvolvimento. A AST busca evitar a criação de modelos funcionais e arquitetônicos dissociados, promover uma abordagem de projeto dirigido por modelos, e, além disso, proporcionar benefícios como independência de plataforma, níveis de abstração mais altos, e a reutilização de informações durante o processo de desenvolvimento de um CPS.

No escopo da AST, os modelos Simulink e AADL são considerados complementares. Ou seja, Simulink permite a captura, a modelagem e a simulação dos requisitos funcionais do sistema de controle, e a AADL suporta refinar o projeto representado pelo modelo funcional Simulink em termos de arquitetura e comportamento. A combinação das duas notações permite um processo combinado que mescla as vantagens da modelagem de alto nível, voltada para a compreensão das funcionalidades do CPS na fase inicial do processo de desenvolvimento, com as vantagens da modelagem de baixo nível voltada para análises, verificações, validação da plataforma de execução, e implementação do código-fonte completo da aplicação (funcionalidades+arquitetura).

A AST explora a capacidade que o Simulink possui de refinar e organizar o projeto detalhado de um sistema de controle por meio blocos do tipo *Subsystem*. São estes os blocos que permitem a identificação de aspectos arquiteturais de um sistema de controle, viabilizando assim a geração de componentes de software AADL no modelo de arquitetura do sistema.

O *plugin* AS2T foi implementado como prova de conceito para verificar o comportamento dos mapeamentos propostos pela AST quando automatizados. Seu funcionamento apresentou resultados satisfatórios durante os experimentos realizados. Mesmo que o modelo AADL gerado pelo *plugin* AS2T seja considerado preliminar, grande parte do trabalho manual empregado para especificar os componentes de software do modelo de arquitetura do sistema é realizado de forma automática. Além disso, o modelo AADL textual gerado pelo *plugin* AS2T pode ser considerado legível, fácil de editar e mais completo, se comparado com um modelo AADL textual equivalente gerado por uma ferramenta que gera o código AADL textual a partir de especificações gráficas, chamada ADELE.

O modelo AADL gerado pelo plugin AS2T pode ser diretamente instanciado, as portas do tipo *data port* e *event data port* dos componentes AADL possuem seus dados devidamente especificados, e as *threads* que o compõem possuem chamadas de subprogramas. Além do mais, os tipos de dados suportados pelas portas dos componentes AADL, herdados do modelo funcional Simulink, bem como os subprogramas que são chamados pelas threads, ficam armazenados em arquivos .aadl diferentes do modelo AADL raiz, o que é interessante para organizar sistemas complexos.

Embora a AST tenha explorado especificamente a transformação de modelos funcionais Simulink em modelos de arquitetura de software AADL, é possível afimar que a essência da AST pode ser generalizada. Ou seja, a solução proposta também pode ser aplicada com outras ferramentas de modelagem funcional, desde que estas sejam baseadas em diagramas de blocos, como por exemplo, as ferramentas LabView e Scilab.

7.1 APLICABILIDADE E LIMITAÇÕES DA TESE

O processo de transformação de modelos proposto nesta tese, a AST, oferece suporte à realização de uma transição bastante crítica durante o processo de desenvolvimento de um CPS, que é a transformação do projeto preliminar (modelo simulink) no projeto detalhado (especificação AADL). A AST facilita o estabelecimento de uma ligação entre os elementos de um modelo funcional (técnicas de controle) e os elementos do modelo de arquitetura computacional do sistema, contribuindo com a consistência e integração destes modelos durante o processo de desenvolvimento do sistema.

A integração de diferentes modelos e ferramentas durante o processo de desenvolvimento de um CPS fornece a possibilidade de explorar os melhores recursos que cada uma delas oferece, recursos estes voltados a análises, verificações e geração de código. No caso específico da AST, é gerado um modelo de arquitetura AADL apto a ser anotado com informações adicionais e ser submetido a vários tipos de análises e verificações que não estão disponíveis para modelos Simulink, o que contribui diretamente com a concepção de um projeto de arquitetura de sistema mais seguro. Sendo assim, a AST contribui com a integração de aplicativos verificados com arquiteturas devidamente validadas, o que tende a tornar o processo de desenvolvimento de um CPS como um todo mais confiável.

Como limitações da solução proposta, pode-se dizer que: (i) como AADL foi essencialmente concebida para representar a estrutura, e não o comportamento, não foi possível fazer uso completo do poder de expressividade dos diagramas Stateflow, (ii) o projetista ainda precisa inserir informações adicionais no modelo AADL gerado pela AST, para que então tal modelo possa ser submetido a análises e verificações, e (iii) não foi projetado e implementado um mecanismo adicional para garantir a sincronização entre os modelos funcional e arquitetural integrados pela AST.

7.2 CONTRIBUIÇÕES

No transcorrer deste trabalho foram identificadas as seguintes contribuições proporcionadas pelo uso da AST:

1. Promove avanços em relação a trabalhos relacionados, uma vez que abrange o mapeamento de construções estruturais, comportamentais e operacionais (modos de operação).

2. Disponibiliza um metamodelo para modelos Simulink e uma versão simplificada do metamodelo AADL;

3. Acelera a geração do modelo de arquitetura do sistema (em AADL) durante o processo de desenvolvimento de um CPS.

4. Estimula o trabalho em conjunto entre os engenheiros de computação e de controle nas fases iniciais do processo de desenvolvimento de um CPS.

5. Oferece suporte para que os engenheiros de computação extraiam informações arquiteturais, operacionais e comportamentais dos

modelos funcionais, gerados pelos engenheiros de controle, sem que para isso precisem dominar as técnicas de controle e nem tão pouco precisem analisar o código gerado pela ferramenta Simulink;

6. Viabiliza a integração de aspectos funcionais com aspectos arquiteturais durante a concepção de um CPS, diminuindo o desacoplamento entre os dois tipos de modelos através da reutilização de informações durante o processo de desenvolvimento de um CPS;

7. Estimula que aspectos arquitetônicos sejam discutidos - e possivelmente testados - em uma fase inicial do processo de desenvolvimento.

8. Contribui com a integração de aplicativos verificados em arquiteturas validadas tornando o processo de desenvolvimento mais robusto.

9. Gera o sistema raiz do modelo de arquitetura AADL que pode ser diretamente instanciado, composto por componentes *subgram*, chamadas de subprogramas dentro das *threads*, e com os tipos de dados das portas do tipo *data port* e *event data port* devidamente especificados;

10. Diminui a ocorrência de erros de inconsistência na definição das conexões entre as portas dos componentes de software no modelo de arquitetura AADL, e garante a consistência dos dados entre as portas envolvidas nestas conexões. Uma vez que as correspondências entre os elementos do modelo funcional e de arquitetura estão bem definidas.

11. Contribui com a realização de análises e verificações no modelo de arquitetura completo (funcionalidades+arquitetura), contribuindo com a concepção de um modelo arquitetura seguro para CPS;

12. Contribui com a integração de duas ferramentas reconhecidas tanto pela comunidade científica quanto pela indústria durante o processo de desenvolvimento de CPS. Oferecendo suporte para que a ferramenta simulink possa atuar como um *front-end* para gerar modelos AADL.

13. Facilita a integração de modelos Simulink com o ambiente TOPCASED. Ou seja, considerando que ambiente de desenvolvimento

TOPCASED faz uso do OSATE, e que este ainda não suporta a transformação de modelos Simulink para AADL, o plugin AS2T também pode ser utilizado para estender a cadeia de transformação do TOPCASED, e facilitar a integração de modelos Simulink com o ambiente TOPCASED.

7.3 PUBLICAÇÕES RESULTANTES DA TESE

A produção técnica vinculada a esta tese resultou nas seguintes publicações científicas:

1. XXVI Simpósio Brasileiro de Engenharia de Software - SBES 2012 (PASSARINI; FARINES; BECKER, 2012)

   Título do Trabalho: Embedded Systems Design: Solution for Generating AADL Architectural Models from Functional Models in Simulink

2. III Simpósio Brasileiro de Engenharia de Sistemas Computacionais - SBESC 2013 (PASSARINI; FARINES; BECKER, 2013)

   Título do Trabalho: The Assisted Transformation of Models: Supporting Cyber-Physical Systems Design by Extracting Architectural Aspects and Operating Modes from Simulink Functional Models

3. Springer Journal of Design Automation for Embedded Systems - 2014

   The Assisted Transformation of Models: Supporting Cyber-Physical Systems Design by Extracting Architectural Aspects and Operating Modes from Simulink Functional Models. (*até o momento da pubicação deste documento, o referido artigo encontra-se em fase de análise por parte dos revisores*)

7.4 SUGESTÕES PARA TRABALHOS FUTUROS

Na sequência são apresentadas questões relevantes a serem consideradas em trabalhos futuros, cujas investigações contribuirão para aprimorar o tema pesquisado.

1. Projetar e implementar um analisador sintático e semântico para verificar se o modelo funcional importado atende as restrições

de modelagem, validando assim, o modelo de entrada com seu respectivo metamodelo.

2. Projetar e implementar um mecanismo adicional para garantir a sincronização e a compatibilidade entre os modelos funcional e arquitetural integrados pela AST.

3. Estender a solução proposta para que a mesma gere uma plataforma de execução *default*, e que especifique uma ligação dos componentes de software com os componentes da plataforma de execução, facilitando a validação do modelo AADL gerado.

4. Estender a solução proposta para que modelos funcionais gerados por outras ferramentas de modelagem, tais como Scade, LabView e Scilab, também possam ser transformados em modelos de arquitetura AADL.

5. Explorar a aplicação de padrões de projeto durante o processo de transformação de modelos proposto.

6. Considerando que a cadeia de verificação do TopCased realiza transformações sucessivas no modelo AADL e gera um modelo autômato equivalente, que pode ser verificado. A AST pode estender a cadeia de verificação existente e, assim, facilitar a integração de modelos Simulink com o ambiente de desenvolvimento TopCased. Portanto, como trabalhos futuros pretende-se sugerir a integração da AST com o ambiente TopCased.

REFERÊNCIAS

ACHILLEOS, A.; GEORGALAS, N.; YANG, K. An open source domain-specific tools framework to support model driven development of oss. In: SPRINGER. *Model Driven Architecture-Foundations and Applications*. [S.l.], 2007. p. 1–16.

ASSOCIATION, M. *Modelica® - A Unified Object-OrientedLanguage for Systems Modeling - Language Specification Version 3.3*. [S.l.], 2012. <https://www.modelica.org/documents/ModelicaSpec33.pdf>.

(AVSI), A. V. S. I. *The System Architecture Virtual Integration Program*. Disponível na Web: http://savi.avsi.aero/: [s.n.], 2010.

BAHETI, K.; GILL, H. Cyber-Physical Systems. *ieeecss.org*, v. 42, n. 3, p. 88–89, mar. 2009. ISSN 0018-9162.

BAHETI, R.; GILL, H. Cyber-physical systems. *The Impact of Control Technology*, p. 161–166, 2011.

BAUDRY, B. et al. Model transformation testing challenges. In: CITESEER. *Proceedings of IMDT workshop in conjunction with ECMDA*. [S.l.], 2006.

BERTHOMIEU, B. et al. Formal verification of aadl models with fiacre and tina. In: *5th International Congress and exhibition ERTS2*. [S.l.: s.n.], 2010.

BERTHOMIEU*, B.; RIBET, P.-O.; VERNADAT, F. The tool tina–construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, Taylor & Francis, v. 42, n. 14, p. 2741–2756, 2004.

BÉZIVIN, J. On the unification power of models. *Software and Systems Modeling*, Springer, v. 4, n. 2, p. 171–188, 2005.

BOSTRÖM, P. et al. *An approach to contract-based verification of Simulink models*. [S.l.], 2010.

BUDINSKY, F. *Eclipse modeling framework: a developer's guide*. [S.l.]: Addison-Wesley Professional, 2004.

CAMPBELL, S. L.; CHANCELIER, J.-P.; NIKOUKHAH, R. *Modeling and Simulation in SCILAB*. [S.l.]: Springer, 2006.

CESAR. The Project CESAR - Cost-eficient methods and processes for safety relevant embedded systems. In: . [s.n.], 2010. <http://www.cesarproject.eu.>.

CETINKAYA, D.; VERBRAECK, A. Metamodeling and model transformations in modeling and simulation. In: *Simulation Conference (WSC), Proceedings of the 2011 Winter*. [S.l.: s.n.], 2011. p. 3043–3053. ISSN 0891-7736.

CHKOURI, M.; BOZGA, M. Prototyping of distributed embedded systems using aadl. *ACESMB 2009*, p. 65, 2009.

CINEL, C. de Formação Profissional da I. E. *Introdução ao LabVIEW*. Disponível em http://www.cinelformacao.com/labview/files/ud1/introd_lv.pdf: [s.n.], 2004.

CORPORATION, N. I. *LabVIEW*. Disponível em http://www.ni.com/labview/pt/: [s.n.], 2013.

CORREA, T. et al. Verification Based Development Process for Embedded Systems. *ETRS - Embedded Real Time Software and Systems*, Toulouse- France, 2010.

CRETTAZ, V. et al. Integrating the concernbase approach with sadl. In: *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. [S.l.]: Springer, 2001. p. 166–181.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, IBM, v. 45, n. 3, p. 621–645, 2006.

DEHAYNI, M. et al. Some model transformation approaches: a qualitative critical review. *Journal of Applied Sciences Research*, v. 5, n. 11, p. 1957–1965, 2009.

DELANGE, J. et al. An mde-based process for the design, implementation and validation of safety-critical systems. *Engineering of Complex Computer Systems (ICECCS), 15th IEEE International Conference on*, p. 319–324, 2010.

EHRIG, H.; ERMEL, C. Semantical correctness and completeness of model transformations using graph and rule transformation. In: *Graph Transformations*. [S.l.]: Springer, 2008. p. 194–210.

ELIPSE. *Eclipse Modeling Framework*. http://www.eclipse.org/emf, 2012. <http://www.eclipse.org/emf>.

ESTEC-ESA, E. *Design complex systems using multiple modeling tools with the ASSERT toolchain*. [S.l.], 2008. <http://www.semantix.gr/assert/Leaflet.pdf>.

FARAIL, P. et al. The topcased project: a toolkit in open source for critical aeronautic systems design. *Embedded Real Time Software (ERTS)*, 2006.

FEILER, P. *AE AADL V2: An Overview*. 2010. <https://wiki.sei.cmu.edu/aadl/images/7/73/AADLV2Overview-AADLUserDay-Feb2010.pdf>.

FEILER, P. H.; GLUCH, D. P. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. [S.l.]: Addison-Wesley, 2012.

FOUNDATION, T. E. *ATL*. Disponível em http://www.eclipse.org/atl/: [s.n.].

FRITZSON, P. *Principles of object-oriented modeling and simulation with Modelica 2.1*. [S.l.]: John Wiley & Sons, 2010.

GONCALVES, F. et al. A model-based design methodology for cyber-physical systems. In: *3rd Workshop Cyber-Physical (CyPhy 2013)*. [S.l.: s.n.], 2013.

HAREL, D.; RUMPE, B. Meaningful modeling: what's the semantics of. *Computer*, IEEE, v. 37, n. 10, p. 64–72, 2004.

IME. *Integrated Modeling Environment*. [S.l.]: http://www.emmeskay.com/tools/ime, 2011.

INRIA. *INRIA EXPRESSO Team. Polychrony*. [S.l.]: http://www.irisa.fr/espresso/Polychrony., 2010.

JENSEN, J.; CHANG, D.; LEE, E. A model-based design methodology for cyber-physical systems. In: *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*. [S.l.: s.n.], 2011. p. 1666 –1671.

KIM, K. Challenges and future directions of cyber-physical system software. In: IEEE. *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*. [S.l.], 2010. p. 10–13.

KLEPPE, A.; WARMER, J.; BAST, W. *MDA explained: the model driven architecture: practice and promise*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2003.

LASNIER, G. et al. An implementation of the behavior annex in the aadl-toolset osate2. In: IEEE. *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*. [S.l.], 2011. p. 332–337.

LASNIER, G. et al. Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In: *Reliable Software Technologies–Ada-Europe 2009*. [S.l.]: Springer, 2009. p. 237–250.

LEE, E. Cyber physical systems: Design challenges. In: IEEE. *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. [S.l.], 2008. p. 363–369.

LEGROS, E. et al. Mate - a model analysis and transformation environment for matlab simulink. In: *Model-Based Engineering of Embedded Real-Time Systems'07*. [S.l.: s.n.], 2007. p. 323–328.

LEGUERNIC, P. et al. Programming real-time applications with signal. *Proceedings of the IEEE*, IEEE, v. 79, n. 9, p. 1321–1336, 1991.

LINA, A. G.; NANTES, I. *ATL User Manual - v0.7*. [S.l.], 2006. <http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[0.7]ṗdf>.

LOPES, D. et al. Mapping specification in mda: From theory to practice. *Interoperability of Enterprise Software and Applications*, Springer, p. 253–264, 2006.

LUCKHAM, D. C. et al. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, IEEE, v. 21, n. 4, p. 336–354, 1995.

MA, L.; XIA, F.; PENG, Z. Integrated design and implementation of embedded control systems with scilab. *Sensors*, Molecular Diversity Preservation International, v. 8, n. 9, p. 5501–5515, 2008.

MA, Y. *Compositional modeling of globally asynchronous locally synchronous (GALS) architectures in a polychronous model of computation*. Tese (Doutorado) — Rennes 1, 2010.

MAGEE, J.; KRAMER, J. Dynamic structure in software architectures. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 1996. v. 21, n. 6, p. 3–14.

MAIA, N. *ODYSSEY-MDA: uma abordagem para a transformação de modelos*. Tese (Doutorado) — UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2006.

MARWEDEL, P. *Embedded system design*. [S.l.]: Springer, 2011.

MATHWORKS, T. *Using Simulink*. 2011. <http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf>.

MAZZINI, S.; PURI, S.; VARDANEGA, T. An mde methodology for the development of high-integrity real-time systems. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.], 2009. p. 1154–1159.

MEDVIDOVIC, N. et al. Using object-oriented typing to support architectural design in the c2 style. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 1996. v. 21, n. 6, p. 24–32.

MEDVIDOVIC, N.; ROSENBLUM, D. S.; TAYLOR, R. N. A language and environment for architecture-based software development and evolution. In: IEEE. *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. [S.l.], 1999. p. 44–53.

MELLOR, S. et al. *MDA distilled*. [S.l.]: Addison Wesley Longman Publishing Co., Inc., 2004.

MENS, T.; GORP, P. V. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 152, p. 125–142, 2006.

MISHRA, P.; DUTT, N. Architecture description languages for programmable embedded systems. *IEE Proceedings-Computers and Digital Techniques*, IET, v. 152, n. 3, p. 285–297, 2005.

MORICONI, M.; RIEMENSCHNEIDER, R. A. *Introduction to SADL 1.0: A language for specifying software architecture hierarchies*. [S.l.], 1997.

NEEMA, S. *A Simulink and Stateflow Data Model*. [S.l.], 2001.

OMG. *MDA Guide Version 1.0*. Disponível em http://www.omg.org/mda/mda_files/MDA _Guide_Version1-0.pdf.: [s.n.], 2003.

OMG. *Meta-Object Facility (MOF), Version 2.0,*. Disponível em http://www.omg.org/spec/MOF/2.0/: [s.n.], 2006. <http://www.omg.org/spec/MOF/2.0/>.

OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification,*. Disponível em http://www.omg.org/spec/QVT/1.0/: [s.n.], 2008. <http://www.omg.org/spec/QVT/1.0/>.

P. Feiler and J. Hudak., D. G. *The architecture analysis & design language (AADL): An introduction*. [S.l.], 2006. <http://www.sei.cmu.edu/library/abstracts/reports/06tn011.cfm>.

PASSARINI, R.; FARINES, J.-M. A.; BECKER, L. B. Embedded systems design: Solution for generating aadl architectural models from functional models in simulink. In: IEEE. *Software Engineering (SBES), 2012 26th Brazilian Symposium on*. [S.l.], 2012. p. 41–50.

PASSARINI, R.; FARINES, J.-M. A.; BECKER, L. B. The assisted transformation of models: Supporting cyber-physical systems design by extracting architectural aspects and operating modes from simulink functional models. In: IEEE. *Brazilian Symposium on Computer System Engineering (SBESC), 2013 3th Brazilian Symposium on*. [S.l.], 2013.

RAGHAV, G.; GOPALSWAMY, S. *Extraction of Simulink to AADL*. Disponível em https://wiki.sei.cmu.edu/aadl/images/6/69/SimulinkToAADL-112009.pdf: [s.n.], 2009.

RAGHAV, G. et al. *Generation of AADL Architecture Consistent Work Products: Simulink Behavioral Models, and Distributed Embedded Software using OCARINA*. Disponível em http://www.aadl.info/aadl/documents/EmmeskaySimulink-AADL Code Generationpublic.pdf: [s.n.], 2009.

RAGHAV, G. et al. Architecture driven generation of distributed embedded software from functional models. 2009.

RAHM, E.; BERNSTEIN, P. A survey of approaches to automatic schema matching. *the VLDB Journal*, Springer, v. 10, n. 4, p. 334–350, 2001.

RUSCIO, D. D. *Specification of model transformation and weaving in model driven engineering*. Tese (Doutorado) — PhD thesis, Universita degli Studi dell'Aquila (February 2007)http://www.di.univaq.it/diruscio/phdThesis. php, 2007.

SAE. *SAE AADL Meta Model and XML/XMI*. 2006. <http://www.aadl.info/aadl/currentsite/tool/metamod.html>.

SAE. *SAE Architecture Analysis and Design Languagem (AADL) Annex Volume 2: Annex B: Data Modeling Annex, Annex D: Behavior Model Annex and Annex F: ARINC653 Annex*. 2011. <http://standards.sae.org/as5506/2>.

SAE. *SAE Standards: Architecture Analysis e Design Language (AADL), AS5506*. [S.l.]: Society of Automotive Engineers, 2011.

SAQUI-SANNES, P. D.; HUGUES, J. et al. Combining sysml and aadl for the design, validation and implementation of critical systems. *ERTS 2012*, 2012.

SCHMIDT, D. Guest editor's introduction: Model-driven engineering. *Computer*, IEEE, v. 39, n. 2, p. 25–31, 2006.

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

SELIC, B. From model-driven development to model-driven engineering. In: *ECRTS*. [S.l.: s.n.], 2007. p. 3.

SENDALL, S.; KOZACZYNSKI, W. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, IEEE, v. 20, n. 5, p. 42–45, 2003.

SHAW, M. et al. Abstractions for software architecture and tools to support them. *Software Engineering, IEEE Transactions on*, IEEE, v. 21, n. 4, p. 314–335, 1995.

SPRINKLE, J. et al. Metamodelling: State of the art and research challenges. In: *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems*. Berlin, Heidelberg: Springer-Verlag, 2010. (MBEERTS'07), p. 57–76. ISBN 3-642-16276-2, 978-3-642-16276-3. <http://dl.acm.org/citation.cfm?id=1927558.1927563>.

TECHNOLOGIES, E. *Scade*. http://www.esterel-technologies.com, 2011. <http://www.esterel-technologies.com>.

TOOM, A. et al. Gene-auto: An automatic code generator for a safe subset of simulink/stateflow and scicos. In: *In European Conference on Embedded Real-Time Software (ERTS08)*. [S.l.: s.n.], 2008.

WIKIPEDIA. *LabVIEW*. Disponível em http://pt.wikipedia.org/wiki/LabVIEW: [s.n.], 2013.

YU, H. et al. System-level co-simulation of integrated avionics using polychrony. In: ACM. *Proceedings of the 2011 ACM Symposium on Applied Computing*. [S.l.], 2011. p. 354–359.

ZOWGHI, D.; COULIN, C. Requirements elicitation: a survey of techniques, approaches, and tools. In: AURUM, A.; WOHLIN, C. (Ed.). *Engineering and Managing Software Requirements*. [S.l.: s.n.], 2005. p. 19–46. DOI 10.1007/3-540-28244-0_2.