Ministério da
Ciência e Tecnologia

sid.inpe.br/mtc-m19/2010/09.29.18.27-TDI

UMA ABORDAGEM EM ARQUITETURA CONDUZIDA POR MODELOS APLICADA A SOFTWARE EMBARCADO DE TEMPO REAL ESPACIAL

Alessandro Gerlinger Romero

Dissertação de Mestrado do Curso de Pós-Graduação em Engenharia e Tecnologia Espaciais / Gerenciamento de Sistemas Espaciais, orientada pelo Dr. Mauricio Gonçalves Vieira Ferreira, aprovada em 10 de novembro de 2010.

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/38BEQL2 >

INPE
São José dos Campos
2010

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

Ministério da
Ciência e Tecnologia

sid.inpe.br/mtc-m19/2010/09.29.18.27-TDI

UMA ABORDAGEM EM ARQUITETURA CONDUZIDA POR MODELOS APLICADA A SOFTWARE EMBARCADO DE TEMPO REAL ESPACIAL

Alessandro Gerlinger Romero

Dissertação de Mestrado do Curso de Pós-Graduação em Engenharia e Tecnologia Espaciais / Gerenciamento de Sistemas Espaciais, orientada pelo Dr. Mauricio Gonçalves Vieira Ferreira, aprovada em 10 de novembro de 2010.

URL do documento original:
<http://urlib.net/8JMKD3MGP7W/38BEQL2 >

INPE
São José dos Campos
2010

Dados Internacionais de Catalogação na Publicação (CIP)

Romero, Alessandro Gerlinger.
R664a Uma abordagem em arquitetura conduzida por modelos aplicada a software embarcado de tempo real espacial / Alessandro Gerlinger Romero. – São José dos Campos : INPE, 2010.
xxvi+ 176 p. ; (sid.inpe.br/mtc-m19/2010/09.29.18.27-TDI)

Dissertação (Mestrado em Engenharia e Tecnologia Espaciais / Gerenciamento de Sistemas Espaciais) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2010.
Orientador : Dr. Maurício Gonçalves Vieira Ferreira.

1. Engenharia conduzida por modelo. 2. Sistemas de tempo real. 3. Sistemas embarcados. 4. Análise de escalonabilidade. 5. Arquitetura conduzida por modelos. I.Título.

CDU 681.3.06

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de Mestre em

Engenharia e Tecnologia Espaciais/Gerenciamento de Sistemas Espaciais

Dr. Nilson Sant'Anna

Presidente / INPE / SJCampos - SP

Dr. Mauricio Gonçalves Vieira Ferreira

Orientador(a) / INPE / SJCampos - SP

Dr. Marcelo Lopes de Oliveira e Souza

Membro da Banca / INPE / SJCampos - SP

Dr. Walter Abrahão dos Santos

Membro da Banca / INPE / São José dos Campos - SP

Dr. Getulio Akabane

Convidado(a) / UNISANTOS / Santos - SP

Aluno (a): Alessandro Gerlinger Romero

São José dos Campos, 10 de novembro de 2010

“Vim a este mundo não, principalmente, para fazer dele um bom lugar para se viver, mas para viver nele, seja bom ou mau. Um homem não tem que fazer tudo, mas algo, e não é porque não pode fazer tudo que precisa fazer este algo de maneira errada."

Henry David Thoreau, em seu livro “A Desobediência Civi”.

Às mulheres da minha vida: Gabriela, Vanessa e Sheila.

AGRADECIMENTOS

Ao concluir este trabalho, lembro-me de muitos que de alguma forma me ajudaram a chegar até aqui. Agradeço a todos, mas correndo o risco de ser injusto quero destacar alguns nomes.

Gostaria de agradecer de forma especial ao orientador do presente trabalho, Professor Doutor Maurício Gonçalves Vieira Ferreira, pela confiança creditada em mim, por permitir que eu explorasse conteúdos pouco difundidos e por contribuir de forma decisiva no presente trabalho.

Gostaria de agradecer também ao Professor Doutor Marcelo Lopes de Oliveira e Souza, pelas disciplinas ministradas e por me proporcionar uma visão sintética e questionadora nos mais variados tópicos de engenharia.

Agradeço também ao Professor Doutor Getúlio Akabane, que orientou meu trabalho de conclusão de graduação e me apoiou nas várias tentativas de executar um trabalho similar a este concluído agora.

Por fim, quero agradecer a meus familiares. A meus pais, Waldomiro e Sheila, por me lançarem como uma flecha na direção do desconhecido. À minha mulher por ser uma verdadeira companheira. E, de forma muito especial, à pequena Gabriela, que me ajuda todo o dia a sondar o espaço infinito que é uma única vida.

Meus sinceros agradecimentos a todos.

RESUMO

Software de tempo real embarcado é intensamente utilizado em missões espaciais. Entretanto seu desenvolvimento carece de técnicas mais robustas para garantir aderência a requisitos cada vez mais abrangentes e complexos, melhorar sua facilidade para execução de testes e integração, além de atender prazos cada vez menores. Alinhado à Engenharia Conduzida por Arquitetura (do Inglês *Model-Driven Architecture*, MDA) o presente trabalho propõe uma abordagem para descrever sistemas de tempo real embarcados usando especificações através de modelos. Tais modelos descrevem todas as partes do sistema em diferentes níveis de abstração, contemplando tanto requisitos funcionais como requisitos não-funcionais. Eles ainda oferecem capacidade para analisar de forma antecipada propriedades funcionais e não funcionais (escalonabilidade, no presente trabalho).

AN APPROACH TO MODEL-DRIVEN ARCHITECTURE APPLIED TO SPACE EMBEDDED REAL TIME SOFTWARE

ABSTRACT

Real time embedded software are applied on space missions, however its development lacks more robust techniques to accomplish requirements more wide and complex, to improve its easiness for execution of tests and integration, beyond increasing time-to-market. Aligned with Model-Driven Architecture the current work proposes an approach to describe real time embedded software using specifications through models. Such models describe all system parts in different abstraction levels, covering functional requirements and non-functional requirements. They still allow early analysis of functional properties and non-functional properties (schedulability, in current work).

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE SIGLAS E ABREVIATURAS

AADL	*Architecture Analysis and Design Language*
ADL	*Architecture Definition Language*
AOCS	*Attitude and Orbit Control System*
CBSE	*Component-Based Software Engineering*
CIM	*Computational Independent Model*
CPU	*Central Processing Unit*
FUML	*Semantics of a Foundational Subset for Executable UML Models*
GRM	*Generic Resource Modeling*
HRM	*Hardware Resource Modeling*
HLAM	*High-Level Application Modeling*
INPE	Instituto Nacional de Pesquisas Espaciais
M2M	*Model to Model*
M2T	*Model to Text*
MARTE	*UML Profile for Modeling and Analysis of Real-Time Embedded Systems*
MDA	*Model-Driven Architecture*
MDE	*Model-Driven Engineering*
MOF	*Meta Object Facility*
NFP	*Non-Functional Properties*
OBC	*On-Board Computer*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
PDM	*Platform Description Model*
PID	*Proportional Integral Derivative*
PIM	*Platform Independent Model*
PM	*Platform Model*
PMM	Plataforma Multi-Missão
PNAE	Plano Nacional de Atividades Espaciais
PSM	*Platform Specific Model*

QVT	*Query View Transformation*
RMA	*Rate Monotonic Analysis*
RTES	*Real Time and Embedded Systems*
SAM	*Schedulability Analysis Modeling*
SPT	*UML Profile for Schedulability, Performance and Time Specification*
SRM	*Software Resource Modeling*
SysML	*Systems Modeling Language*
UML	*Unified Modeling Language*
VSL	*Value Specification Language*
WCTE	*Worst Case Time Execution*
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*

SUMÁRIO

Pág.

1 INTRODUÇÃO

Software de tempo real embarcado é intensamente utilizado em missões espaciais. Entretanto, seu desenvolvimento carece de técnicas mais robustas para garantir aderência a requisitos cada vez mais abrangentes e complexos, melhorar sua facilidade para execução de testes e integração, e atender prazos cada vez menores.

Nos últimos anos, o uso de modelos precisos para descrever tais arquiteturas e soluções vem chamando a atenção tanto do meio acadêmico quanto do mercado. O presente trabalho pretende usar as principais especificações do OMG (*Object Management Group*) para definir um conjunto de modelos e ferramentas que permitam a definição completa de uma solução, assim como a análise antecipada de suas propriedades não funcionais, neste trabalho escalonabilidade (*schedulability*).

1.1. Motivação do trabalho de pesquisa

A complexidade do software embarcado de tempo real tem crescido muito rapidamente. Tão relevante quanto isso, para o presente trabalho, é que o esforço despendido em cada uma das tarefas do desenvolvimento do software varia muito conforme a complexidade (para fins didáticos assume-se aqui a seguinte simplificação: tamanho é proporcional a complexidade), deslocando-se fortemente tal esforço da codificação para a arquitetura, integração e testes do sistema, isto é indicado na Figura 1.1. Assim, buscar maneiras para aprimorar a forma de codificação não trata totalmente as necessidades atuais.

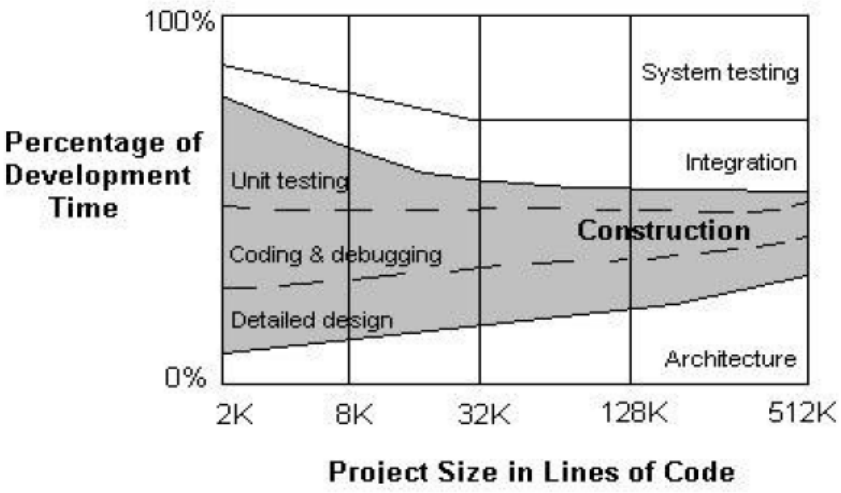

Figura 1.1 - Complexidade do software e esforço necessário.
Fonte: Souza e Trivelato (2003).

Uma alternativa para tratar o problema de forma mais completa é seguir o caminho de outras engenharias (KENNEDY CARTER, 2002), sendo este:

1. Construir modelos precisos;
2. Submeter os modelos a testes rigorosos antes da implementação;
3. Estabelecer uma forma bem definida (e possivelmente automatizada) para o processo de construção;
4. Construir o produto usando componentes reutilizáveis.

O item “1. Construir modelos precisos” declara a necessidade de modelos, ou mais precisamente, determina que modelos devam conduzir todo o processo de engenharia (MDE - *Model-Driven Engineering*).

Uma vez que modelos precisos existam, é possível obter os seguintes benefícios (FEILER e NIZ, 2008):

- Redução de riscos: análise antecipada do sistema e durante o ciclo de vida; análise de impacto; validar premissas no sistema.

- Aumento da confiabilidade: validar modelos para complementar testes de integração; validar premissas do modelo no ambiente operacional;
- Redução de custo: menos problemas de integração; suporte de ferramentas de engenharia; suporte ao ciclo de vida mais simples.

A Tabela 1.1 apresenta casos de sucesso no uso de MDE.

Tabela 1.1 – Casos de sucesso MDE.

Empresa	**Produto**	**Benefícios**
Airbus	A340	Redução de 20 vezes no número de erros Redução de prazo
GE & Lockheed Martin	FADEDC Engine Controls	Redução dos erros 50% de redução do tempo Custo reduzido
Schneider Electric	Nuclear Power Plant Safety Control	Redução de oito vezes no número de erros enquanto a complexidade aumentou quatro vezes
Honeywell Commercial Aviation Systems	Primus Epic Flight Control System	Aumento da produtividade em cinco vezes Não foram localizados erros de código Recebeu certificação FAA

Fonte: adaptada de Feilter e Niz (2008).

Explorando ainda os benefícios do uso da engenharia conduzida por modelos, a Figura 1.2 mostra que a MDE pode contribuir significativamente para aumentar o número de falhas localizadas durante as fases iniciais de um projeto de desenvolvimento de software. Dado que em um modelo clássico de desenvolvimento, testes só são realizados após a existência do código. Entretanto com MDE, testes podem ser realizados já sobre o projeto ou arquitetura do sistema.

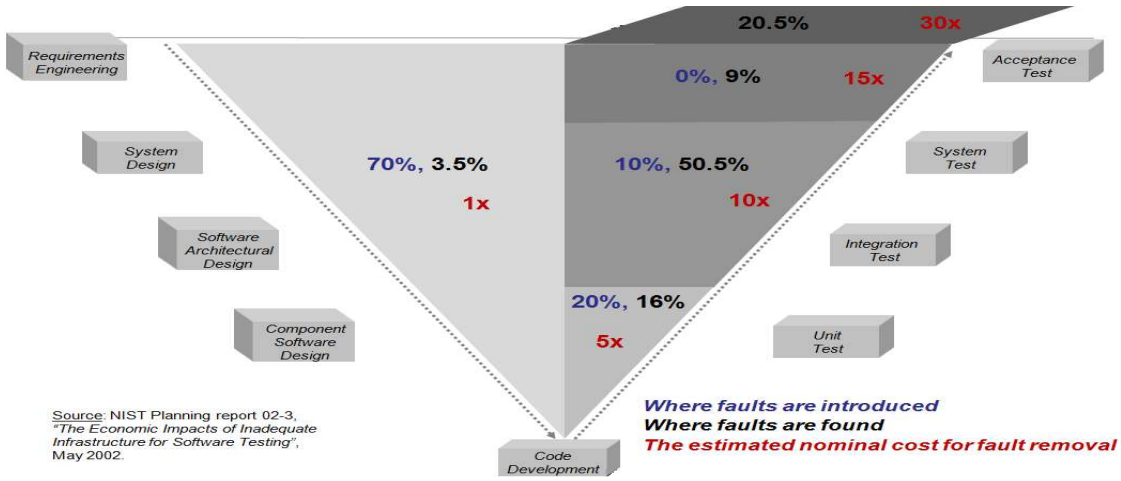

Figura 1.2 – Onde falhas são introduzidas, localizadas e qual o custo delas.
Fonte: Lewis e Feiler (2009).

Ainda em Feiler e Niz (2008) é abordada a questão se a MDE pode ser usada para projetos realmente grandes. Esta pergunta é respondida citando o exemplo do Airbus A380 (o maior avião de passageiros, já construído), onde os seguintes sistemas foram desenvolvidos usando-se MDE: Controle de vôo, piloto automático e outros.

Entretanto uma pesquisa de 2008 (NASS, 2008) com desenvolvedores especializados em software embarcado indica que 43% deles usam alguma linguagem de abstração, como UML, Simulink ou SystemC. Adicionalmente, 55% deles desejam usar um nível maior de abstração, vide Figura 1.2 (NASS, 2008).

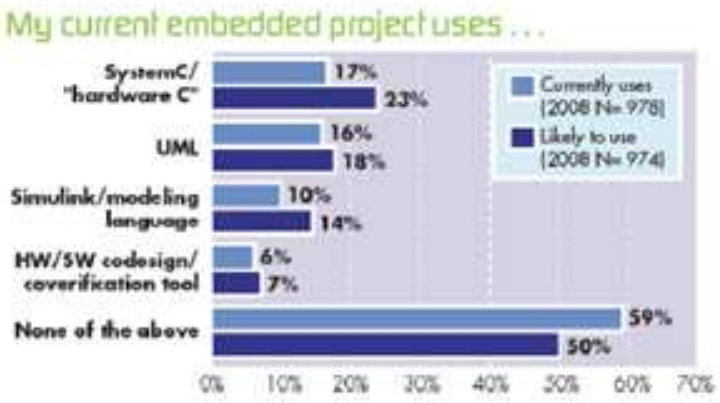

Figura 1.3 – Uso de níveis mais abstratos.
Fonte: adaptada de Nass (2008).

É possível inferir com base nesta pesquisa, que a prática do uso de MDE não é ainda intensivamente disseminada no mercado, em parte devido à imaturidade das especificações e das ferramentas necessárias. Tão importante quanto usar modelos com um nível mais alto de abstração para projetar software, é usar tais modelos para conduzir todo o processo, ou seja, deslocar o foco do código para modelos.

Congressos, como o *Model Based Architecting and Construction of Embedded Systems* ou o *Model Driven Engineering Languages and Systems*, focados especialmente no uso de MDE em sistemas embarcados de tempo real têm ganhado cada vez mais atenção. Isto é mais uma indicação da relevância do tema para a comunidade acadêmica.

Assim a motivação deste trabalho é explorar recentes especificações para viabilizar MDE com todo o seu potencial, contribuindo para o desenvolvimento de novas aplicações embarcadas de tempo real usando-se MDE no INPE e no Brasil.

1.2. Objetivo geral do trabalho de pesquisa

Alinhado à MDE, este trabalho de pesquisa tem como objetivo propor uma abordagem para descrever sistemas de tempo real embarcados usando especificações de mercado através de modelos. Tais modelos descrevem todas as partes do sistema em diferentes níveis de abstração, contemplando tanto requisitos funcionais como requisitos não-funcionais. Eles ainda oferecem capacidade para analisar de forma antecipada propriedades funcionais e não funcionais (escalonabilidade, no presente trabalho).

Uma alternativa promissora para MDE é a MDA (*Model-Driven Architecture*) (OMG, 2003). Ela foi inicialmente proposta para a Engenharia de Software, mas está sendo também explorada na Engenharia de Sistemas (CLOUTIER, 2006). O ponto chave da MDA é a existência de modelos com diferentes níveis de abstração. É objetivo explorar o modelo independente de plataforma (PIM – *Platform Independent Model*) (OMG, 2003) e o modelo específico para uma plataforma (PSM – *Platform Specific Model*) (OMG, 2003).

Para a definição do PIM é usada a especificação fUML (*Semantics of a Foundational Subset for Executable UML Models*) (OMG, 2009c) que define um subconjunto da UML a ser usado para gerar modelos executáveis. Com esta especificação é possível testar funcionalmente o modelo.

Para transformar o PIM em PSM é usada a especificação MOF QVT (*Query/View/Transform*) (OMG, 2008a) gerando um PSM que é instrumentado com o UML *profile* MARTE (*UML Profile for Modeling and Analysis of Real-Time Embedded Systems*) (OMG, 2009a). A partir deste PSM, a especificação MOF M2T (OMG, 2009b) é usada para a transformação do PSM em código Java e também em um modelo AADL (*Architecture Analysis and Design Language*) (FEILER et al., 2006).

O modelo AADL permite que a análise de escalonabilidade seja efetuada sem à necessidade de avaliação do código gerado em Java.

Estas especificações, modelos e transformações serão aplicados em uma avaliação de aplicabilidade a ser realizada com parte do controle de atitude da PMM (Plataforma Multi-Missão) em modo nominal apresentado por Moreira (2006).

1.3. Objetivos específicos do trabalho de pesquisa

Abaixo seguem os objetivos específicos:

- Definir o que deve ser modelado e em que nível (PIM ou PSM), considerando requisitos funcionais e não funcionais;
- Definir o que usar de cada especificação selecionada, sendo elas: fUML, MARTE, MOF, MOF QVT e MOF M2T;
- Criar protótipos das transformações:
    - PIM para o PSM;
    - PSM para AADL;
    - PSM para código Java;
- Usar as definições anteriores na modelagem de um PIM para a avaliação de aplicabilidade considerando parte do subsistema de controle de atitude da PMM, seguindo a modelagem de controle apresentada por Moreira (2006);
- Testar funcionalmente, de modo parcial através de um protótipo, o modelo PIM;
- Analisar a escalonabilidade durante a avaliação de aplicabilidade;
- Gerar um código executável, em Java, durante a avaliação de aplicabilidade;

O capítulo seguinte apresenta uma breve revisão bibliográfica dos principais conceitos e especificações necessários para o presente projeto.

1.4. Organização do texto

Este documento está estruturado da seguinte forma: o Capítulo 2 apresenta a revisão bibliográfica dos itens fundamentais para o desenvolvimento do trabalho, o Capítulo 3 introduz a arquitetura da proposta e o Capítulo 4 detalha a implementação. Em seguida, o Capítulo 5 apresenta os resultados obtidos durante a pesquisa e o Capítulo 6 apresenta as conclusões finais. As referências bibliográficas são listadas no fim deste documento.

2 REVISÃO BIBLIOGRÁFICA

O capítulo corrente revisa os principais conceitos a serem usados no presente documento, assim como no projeto proposto.

Não existe consenso na definição dos termos tempo real e embarcado, entretanto é possível obter na literatura as definições detalhadas a seguir para sistemas embarcados de tempo real, RTES (*real time embedded systems*).

2.1. Sistemas de tempo real

Segundo Stankovic (1988) um sistema de tempo real é aquele onde o comportamento correto do sistema depende tanto da correção lógica dos resultados (*correctness*) quanto do tempo em que tais resultados são produzidos (*timeliness*). Definições similares são encontradas na literatura especializada (KOPETZ, 1997) (FARINES et al., 2000) (BRIAND e ROY, 1999).

Quanto à segurança, sistemas de tempo real podem ser classificados em (FARINES et al., 2000): sistemas críticos de tempo real (*hard real time systems*), aqueles onde uma falha devido ao tempo pode causar perdas catastróficas; e sistemas não críticos de tempo real (*soft real time systems*), aqueles onde uma falha devida ao tempo não compromete seriamente o resultado.

Sistemas críticos de tempo real são freqüentemente encontrados interagindo com o ambiente, como sistemas embarcados de controle (WEHRMEISTER, 2009). Por exemplo, o sistema de controle de atitude e órbita da PMM é um sistema embarcado crítico de tempo real, pois um atraso no controle de atitude pode comprometer a missão espacial.

Sistemas de tempo real possuem algumas importantes características, entre elas destaca-se a previsibilidade (*predictability*). Um sistema de tempo real é dito previsível (*predictable*) se, independente de variações que ocorram no hardware, na carga e/ou em falhas, o comportamento do sistema pode ser antecipado, antes de sua execução (FARINES et al., 2000). Outra característica importante é a escalonabilidade (*schedulability*). Dado que um sistema é formado por um conjunto de tarefas, e que suas tarefas são supostas como funcionalmente corretas, o problema é como garantir, em tempo de projeto, que todas elas irão satisfazer os requisitos de tempo durante a execução. E isto mesmo que tais tarefas tenham prazos (*deadlines*) distintos, relações de precedência, relações de exclusão; e ainda que, concorram pelo mesmo processador. Ou seja, escalonabilidade é uma propriedade não funcional do sistema que determina se este irá atender os requisitos temporais de todas as tarefas, mesmo no pior caso.

O problema de tempo real não é resolvido pelo aumento da velocidade dos computadores. Entretanto a rapidez de cálculo minimiza o tempo médio de execução de um conjunto de tarefas, enquanto que o cálculo em tempo real garante o atendimento dos requisitos temporais de cada tarefa (STANKOVIC, 1988).

2.2. Sistemas embarcados

Sistemas embarcados são geralmente definidos como um conjunto de dispositivos interconectados que contêm hardware e software (OMG, 2009a). Sistemas embarcados são sistemas baseados em computadores instalados em um ambiente com o qual interagem. Um exemplo já citado é o sistema de controle de atitude e órbita de um satélite.

O desenvolvimento de sistemas embarcados implica o projeto de um sistema no qual os recursos são usualmente limitados. Como os recursos são finitos (tamanho da memória, consumo de energia, capacidade de processamento, etc.) o desenvolvimento de sistemas embarcados requer otimização (OMG, 2009a). Além disso, eles devem usualmente ser executados sem intervenção manual, assim todos os erros precisam ser tratados. Outro ponto importante em sistemas embarcados é que eles são caracterizados pela completa ausência ou uma redução muito significativa da interface de usuário. Isto já os coloca à parte de outros tipos mais convencionais de sistemas nos quais a interface com o usuário representa 50% do esforço total de desenvolvimento e codificação do software (PASETTI e BROWN, 2002).

2.3. Sistemas orientados a eventos

Sistemas embarcados são essencialmente orientados a eventos, o que significa que eles continuamente esperam a ocorrência de um evento interno ou externo como um sensor ter uma nova leitura, um botão ser pressionado ou a chegada de um novo pacote de dados. Depois de reconhecido o evento, tais sistemas reagem executando o processamento apropriado que pode incluir manipular um hardware ou gerar outros eventos que disparam componentes de software internos. É por isso que sistemas orientados a eventos são também chamados de sistemas reativos (SAMEK, 2009). Quando o tratamento de um evento é completado, o sistema volta a esperar um próximo evento.

Um exemplo já citado é o sistema de controle de atitude e órbita de um satélite, que reage a novas leituras dos sensores ou a telecomandos recebidos do OBC (*On-Board Computer*).

2.4. Sistemas de supervisão e controle

Sistemas de supervisão e controle são usualmente dedicados ao gerenciamento da execução de um processo ou de um objeto do mundo físico (OMG, 2009a). Uma definição mais aprofundada de sistemas deste tipo pode ser encontrada na literatura especializada (DORF e BISHOP, 2005) (OGATA, 1997).

Neste tipo de sistema é possível distinguir os seguintes componentes: Operador, Sistema Computacional de Controle e o Sistema a Controlar. O Sistema a Controlar e o Operador são considerados como ambiente do sistema computacional. A interação entre as partes ocorre através de interfaces de instrumentação (compostas de sensores e atuadores) e da interface com operador (FARINES et al., 2000). A Figura 2.1 mostra estes componentes em conjunto com elementos clássicos de um diagrama de controle.

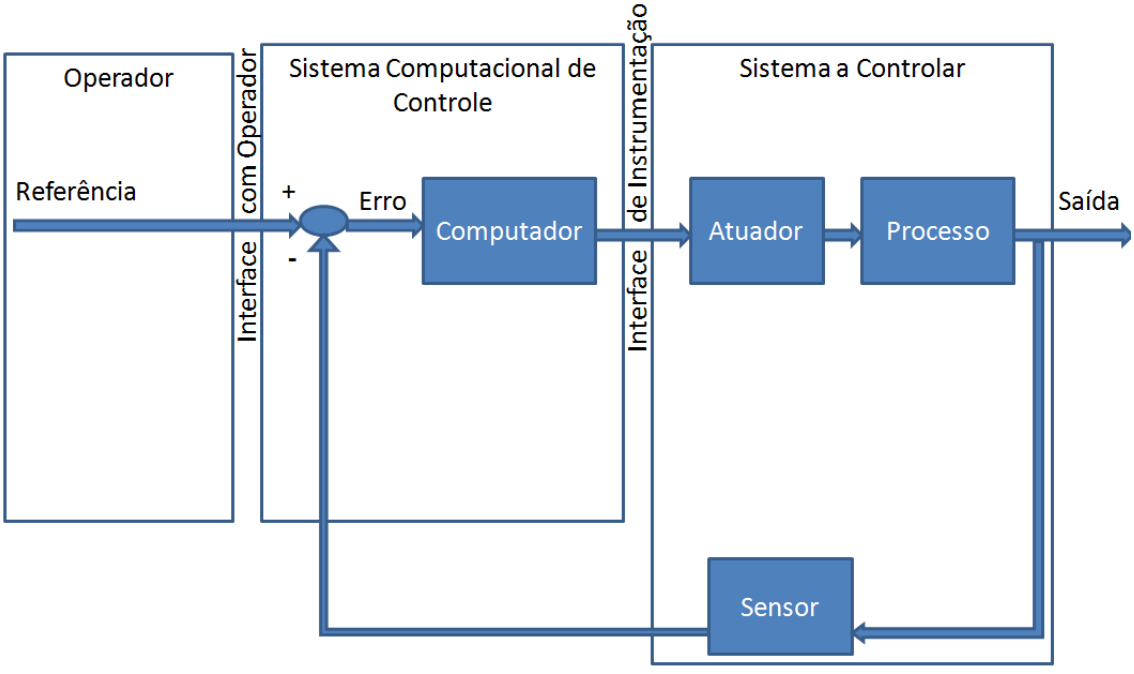

Figura 2.1 – Sistema de supervisão e controle de tempo real.
Fonte: adaptada de Farines et al. (2000).

2.4.1. Controle

Antes de seguir com a revisão bibliográfica, é preciso ao menos citar a Teoria de Controle. Um exemplo de literatura consagrada sobre o tema pode ser encontrada em Ogata (1997).

O trabalho de Moreira (2006) usa intensamente o conceito de função de transferência, que segundo Ogata (1997) é definida como a razão entre a transformada de Laplace da função de saída e a transformada de Laplace da função de entrada sobre a premissa de condições iniciais iguais a zero. Ogata (1997) destaca que a aplicabilidade do conceito de função de transferência é limitada a sistemas descritos através de equações diferenciais lineares e invariantes no tempo; e ainda que a função de transferência é uma propriedade do próprio sistema, independente da magnitude ou da natureza da função de entrada. Estes conceitos básicos de controle são importantes, pois serão usados na seqüencia do trabalho.

Ainda segundo Moreira (2006), mesmo havendo técnicas sofisticadas de controle, o algoritmo mais utilizado para controle de atitude de veículos espaciais é o algoritmo proporcional, integral e derivativo (PID, *Proportional-plus-Integral-plus-Derivative*).

Para terminar esta parte de conceitos de sistemas embarcados de tempo real é possível concluir que o sistema selecionado para a avaliação de aplicabilidade no presente trabalho, o sistema de controle de atitude e órbita da PMM, se encaixa nas quatro classificações de sistemas abordadas: sistema de tempo real, sistema embarcado, sistema orientado a evento e sistema de supervisão e controle.

2.5. Abordagens para aplicação de MDE

Uma abordagem difundida para a Engenharia Conduzida por Modelos é aquela onde engenheiros de controle usam ferramentas como Matlab (MATHWORKS, 2010a) ou MATRIXx (MATHWORKS, 2010b) para o projeto de sistemas de supervisão e controle aplicando abstrações da teoria de controle, como: funções de transferência, equações no espaço de estados, controladores PID, etc...

Ambientes como o Matlab e o MATRIXx fazem um ótimo trabalho modelando e implementando código para os algoritmos de controle, entretanto isto representa apenas um pequeno subconjunto do software em aplicações de controle complexas, por exemplo, em um sistema de controle de atitude e órbita de um satélite isto representa de 20 a 30% do código (PASETTI e BROWN, 2002). Além disso, Pasetti e Brown (2002) ainda salientam que estes ambientes não facilitam o projeto da arquitetura do software gerando códigos difíceis de entender e com uma estrutura obscura.

É importante destacar que ambientes como esses são importantíssimos para o desenvolvimento de sistemas de supervisão e controle, entretanto não se deve creditar somente a eles a tarefa da modelagem do software.

Abordando mais especificamente MDE para software, Flint e Boughton (2003) classificam os métodos para desenvolvimento de software em dois tipos:

- Elaborativo – a abordagem elaborativa começa durante a análise criando-se modelos de software em um nível alto de abstração, omitindo-se detalhes de projeto e implementação. Durante as fases de projeto estes modelos são elaborados (ou refinados) para incluir informações sobre o projeto e a implementação, mascarando-se a distinção entre especificação de requisitos e descrição do projeto.

- Baseado em transformação – Modelos produzidos durante as atividades de análise especificam dados, estado e comportamento necessários para cada aspecto do sistema independente da implementação. A informação destes modelos são então traduzidas e entrelaçadas (*woven*) com detalhes de arquitetura e modelos de implementação para formar o código e outros artefatos.

A abordagem baseada em transformação para desenvolvimento de software foi proposta inicialmente por Sally Shlaer e Stephen Mellor na década de 80 (FLINT e BOUGHTHON, 2003). Com a adoção de algumas destas idéias pelo OMG (*Object Management Group*) na MDA estes tipos de métodos têm sido refinados e atualizados. Ferramentas comerciais, como a da fornecedora Kennedy Carter (KENNEDY CARTER, 2002), permitem a geração completa de sistemas a partir de modelos UML.

Tais métodos, aliados a especificações do OMG, representam um nível mais alto de abstração no desenvolvimento de software. A própria história da engenharia de software sustenta que em breve estaremos produzindo software em um nível mais alto de abstração (PRESSMAN, 2006).

2.6. Abordagens para implementação de sistemas de tempo real

Baseada na maneira de tratar a simultaneidade existem duas abordagens diferentes para a implementação de sistemas de tempo real, amplamente discutidas na literatura, sendo elas (FARINES et al., 2000):

- Abordagem Assíncrona - trata a ocorrência e a percepção de eventos independentes numa ordem arbitrária, mas não simultânea. Ela está fundamentada no tratamento explícito da simultaneidade e do tempo de uma aplicação em execução; a questão do escalonamento de tempo real, ou escalonabilidade, é o ponto principal do estudo da previsibilidade dos sistemas de tempo real que seguem esta abordagem.

As linguagens Ada e Java podem ser consideradas como linguagens assíncronas (FARINES et al., 2000).

- Abordagem Síncrona - assume a hipótese de que os cálculos e as comunicações não levam tempo. Nela a simultaneidade é resolvida sem o entrelaçamento de tarefas e o tempo não é tratado de maneira explícita. Esta abordagem é considerada como orientada ao comportamento de uma aplicação e à sua verificação. As linguagens Esterel e Lustre são consideradas linguagens síncronas (FARINES et al., 2000).

2.7. Abordagens para modelagem de sistemas orientados a eventos

A abordagem dominante para a modelagem de software é o paradigma "evento-ação" (*Event-Action*), onde os eventos são diretamente mapeados para o código que deve ser executado em resposta dos mesmos (SAMEK, 2009). Isto fica claro quando se observa que o Diagrama de Seqüência da UML (*Unified Modeling Language*) é o mais utilizado para a definição de comportamento de componentes e/ou aplicações. Um exemplo do emprego deste paradigma é o trabalho de Wehrmeister (2009), que usa os diagramas de seqüência como principal fonte para a modelagem de comportamento.

Com este paradigma, uma aplicação não possui a noção de um único modo de operação, mas em vez disso, um conjunto de condições extremamente acopladas e repetitivas determinadas por valores de variáveis globais definem o estado corrente da mesma (SAMEK, 2009). Tal abordagem parece funcionar bem inicialmente, entretanto ela não escala quando o software cresce em complexidade.

Nesta abordagem problemas são facilmente inseridos no software, devido às seguintes razões (SAMEK, 2009):

- Sempre leva a uma lógica condicional extensa;

- As avaliações de expressões complexas são necessárias em muitos pontos;
- Mudar de um modo para outro requer a alteração de muitas variáveis, o que facilmente pode levar a inconsistências.

Assim, é esperado que um evento sozinho não determine as ações a serem executadas em resposta a ele. O contexto corrente tem ao menos a mesma importância. Entretanto, o paradigma “Evento-Ação” reconhece somente a dependência do tipo de evento e deixa a manipulação do contexto para técnicas manuais que facilmente comprometem o software (SAMEK, 2009).

2.8. Especificações OMG

Um problema fundamental na maioria das propostas e ferramentas de MDE é a falta da utilização de padrões consagrados e amplamente aceitos para a especificação do modelo (SILVA, 2005). Para evitar este problema o corrente trabalho adere completamente às especificações mais recentes do OMG.

2.8.1. MDA

A MDA é uma iniciativa do OMG que promove a produtividade no desenvolvimento de software, assim como a portabilidade e a manutenabilidade de sistemas de software (OMG, 2003) (FLINT e BOUGHTHON, 2003). Ela promove estes benefícios explorando o bem estabelecido princípio de separar a especificação do software de seu projeto e implementação.

MDA é uma abordagem para desenvolvimento de software, cujo objetivo é usar uma linguagem de modelagem como uma linguagem de programação de alto nível, e não apenas como uma linguagem de *design* (WANG, 2005). UML executável é a meta final da MDA (WANG, 2005). Para realizar esta meta, o OMG criou uma série de especificações, destacando-se: UML (OMG, 2009b),

MOF (OMG, 2006), MOF QVT (OMG, 2008a), MOF M2T (OMG, 2008b) e fUML (OMG, 2009c). Usar MDA implica que modelos são o ponto central do desenvolvimento de software, o que está alinhado com a definição apresentada de Engenharia Conduzida por Modelos.

MDA define:

- Modelos independentes de computador (CIM - *Computation Independent Model*) são descrições, essencialmente no formato texto (WANG, 2005), do sistema na visão do especialista de domínio (OMG, 2003);
- Modelos independentes de plataforma (PIM), que capturam os requisitos de software completamente livres de qualquer preocupação de projeto ou implementação. É o primeiro modelo formal definido na MDA (OMG, 2003);
- Modelos específicos para plataforma (PSM) que representam o projeto e a implementação do software (OMG, 2003).

PIMs são transformados em um ou mais PSMs de acordo com regras de transformações bem definidas. PSMs produzidos são novamente transformados em PSMs mais específicos e eventualmente em código. A Figura 2.2 ilustra um dos possíveis processos de transformação usando MDA. Wang (2005) destaca que as transformações automatizadas são uma das grandes revoluções da MDA, e complementa que a transformação de PIM para PSM é o passo mais difícil da MDA.

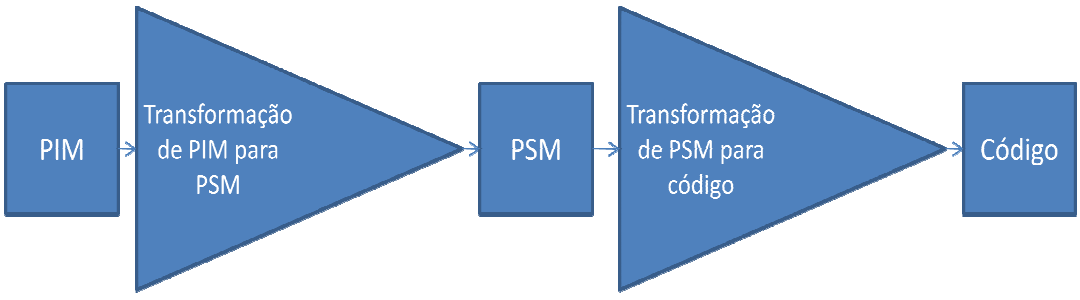

Figura 2.2 – Abordagem MDA.

Fonte: adaptada de Silva (2005).

Alguns fornecedores (KENNEDY CARTER, 2002) de ferramentas argumentam que com um modelo PIM completo (definindo comportamento, estado e estrutura) é possível desconsiderar o modelo PSM e gerar a partir do PIM o código final da aplicação usando-se uma transformação focada em uma plataforma. Nesta abordagem os especialistas do sistema a ser modelado centram seus esforços na modelagem e teste de um modelo PIM completo. Nela, os especialistas em software focam seus esforços na modelagem e teste de transformações de um PIM qualquer para uma determinada plataforma, usando padrões de projeto de software (*design patterns*), características da plataforma e de seus componentes. Este tipo de abordagem viabiliza o reuso do modelo PIM, assim como da transformação desde que utilizada a mesma plataforma como destino. Entretanto um ponto fraco desta abordagem é que, ao se suprimir o modelo PSM é impossível avaliar propriedades não funcionais do sistema (*NFP – Non Functional Properties*), como a escalonabilidade, sem avaliação exaustiva do código gerado. Está é a abordagem chamada por Mraidha (2006) de mapeamento explícito. Mais detalhes são apresentados na seção referente à especificação MOF M2T.

Sobre os benefícios da MDA, cabe ao menos explorar três deles:

- Produtividade – existe um grupo trabalhando na modelagem do negócio no PIM, enquanto outro grupo trabalha na definição das transformações para o PSM. Isto gera uma separação de responsabilidade clara, e permite o paralelismo, e ainda mais, quando uma transformação para PSM for finalizada, ela pode ser reutilizada um número muito grande de vezes.
- Portabilidade – como o modelo PIM é o único artefato que contem a modelagem do negócio e ele é independente de plataforma, ao se selecionar outra plataforma apenas a transformação para PSM deve ser reavaliada;
- Manutenabilidade – como o modelo PIM é o único artefato que contém a modelagem de negócio, ele possui, ao mesmo tempo, todos os

elementos necessários (do ponto de vista do negócio) para geração do código e toda documentação necessária.

2.8.2. UML

O objetivo da UML (*Unified Modeling Language*) é prover ferramentas para análise, projeto e implementação de sistemas baseados em software assim como para modelagem de negócios e processos similares (OMG, 2009b). As versões iniciais da UML foram baseadas em três métodos líderes para orientação a objetos, sendo eles: *Booch*, *Object Modeling Technique* e *Object-Oriented Software Engineering*. A estes foram incorporados melhores práticas de linguagens de modelagem, programação orientada a objetos e linguagens para descrição de arquiteturas (ADLs – A*rchitectural Description Languages*).

Embora os modelos PIM e PSM possam ser representados em qualquer linguagem suportada, o OMG aponta a UML como a linguagem ideal para a construção destes modelos (SILVA, 2005).

Um elemento que favorece a adoção da linguagem UML para a construção de modelos PIM na arquitetura MDA é a sua característica extensível (SILVA, 2005). Essa extensibilidade é fornecida pelos UMLs *Profiles*. Através de UML *Profiles* é possível estender a linguagem UML para algum aspecto não contemplado na mesma e que seja de interesse num contexto em particular.

Existem duas premissas fundamentais da UML que merecem ser destacadas (OMG, 2009b), pois têm impacto direto no corrente trabalho. São elas:

1. Todo o comportamento em um sistema modelado é, em última instância, causado por ações executadas por objetos ativos, aqueles que possuem sua própria linha de execução (*thread*) de controle;
2. Somente modela comportamentos orientados a eventos. Entretanto, a UML não define o tempo entre eventos, o que pode ser definido

como tão pequeno quanto necessário para uma aplicação, por exemplo, quando simula comportamentos contínuos.

2.8.3. MOF

MDA exige a utilização de modelos escritos numa linguagem bem-definida, ou seja, que possa ser interpretada não só por seres humanos, mas também por computadores. Uma linguagem bem-definida é aquela que obedece a um conjunto de regras formais, possui uma forma (sintaxe) e um significado (semântica). MOF (*Meta Object Facility*) é uma linguagem para descrição de meta-modelos que, por sua vez, irão definir regras sobre como os modelos devem ser escritos para serem considerados bem definidos (SILVA, 2005).

A especificação da infra-estrutura da UML (OMG, 2009d) define quatro camadas hierárquicas de meta-modelo, sendo elas:

- M3 – meta-meta-modelo, onde são definidos elementos básicos que permitam a definição da próxima camada hierárquica;
- M2 – meta-modelo, onde são definidos os elementos a serem usados para um determinado tipo de modelo. Um exemplo de M2 é a própria UML;
- M1 – modelo, onde são definidos os elementos do sistema a ser modelado, por exemplo, Carro;
- M0 – instancias, onde são instanciados os elementos do nível M1, por exemplo, fusca é uma instância, na camada M0, da classe Carro da camada M1.

MOF tem contribuído significativamente para alguns dos princípios fundamentais da MDA. Com a introdução do conceito de meta-modelos é possível definir mapeamentos de PIMs para PSMs (OMG, 2006).

O uso de MOF em transformações pode ser classificado em dois tipos:

- Transformações de modelo para modelo (*model to model*, m2m), onde a especificação MOF *Query/View/Transformation* define uma linguagem para a realização de tais transformações (OMG, 2008a).
- Transformações de modelo para texto (*model to text*, m2t), onde a especificação MOF *Model to Text Transformation Language* define uma linguagem para a realização de tais transformações (OMG, 2008b).

2.8.4. MOF QVT

A especificação MOF QVT (*Query/View/Transformation*) é focada na definição de linguagens para transformações do tipo modelo para modelo (OMG, 2008a). Como a MDA tem a transformação de modelos, em especial a transformação de PIM para PSM, as transformações de modelo para modelo possuem um papel chave na MDA.

O objetivo de transformações do tipo modelo para modelo é gerar um novo modelo a partir de um modelo de origem, seguindo regras formalmente definidas (MRAIDHA, 2006), para:

- *Query* – é usada para extrair um subconjunto de interesse do modelo fonte (MRAIDHA, 2006). Uma *query* é uma expressão que é avaliada sobre o modelo de origem, o resultado dela é uma ou mais instâncias de tipos definidos no meta-modelo de origem. Um exemplo é localizar todas as classes que possuem mais de um pai (MRAIDHA, 2006).
- *Transformation* – é usada para gerar um novo modelo, com algum nível de equivalência, a partir do modelo de origem. A equivalência depende do propósito da transformação, pode ser uma equivalência focada em níveis de abstração diferentes, ou em formas de representação distinta. Pode ser bidirecional ou unidirecional. Um exemplo é a transformação de PIM para PSM.

- *View* – segundo Mraidha (2006) é um caso especial de transformação, onde o modelo de origem é representado a partir de um ponto de vista (*viewpoint*) particular.

A especificação (OMG, 2008a) define três linguagens diferentes, complementares e estaticamente tipificadas para transformações:

- *Core* – é uma linguagem declarativa baseada na localização de padrões sobre um conjunto de variáveis e na avaliação de condições sobre tais variáveis contra um conjunto de modelos;
- *Relations* – é uma linguagem declarativa que permite a especificação de um conjunto de relações entre modelos MOF de forma declarativa, ou seja, permite transformações bi-direcionais;
- *Operational* – é uma linguagem imperativa similar às linguagens de programação procedurais, mas ela também contém construções para manipular elementos MOF. Esta linguagem foi projetada, segundo (DVORAK, 2008), para transformações que precisam gerar modelos destino com uma estrutura complexa, nos casos onde não existe uma correspondência direta entre um elemento individual no modelo de origem e no modelo de destino, o que dificulta a descrição do relacionamento de forma declarativa.

As três linguagens são extremamente baseadas na linguagem declarativa OCL (*Object Constraint Language*). Além destas três linguagens, ainda existe uma forma definida na especificação para chamar algoritmos complexos codificados em qualquer linguagem de programação com acesso aos elementos MOF, chamada de *Blackbox Library*.

As duas linguagens declarativas são semanticamente equivalentes, entretanto a especificação declara que a *Relations* é amigável ao usuário enquanto que a Core é de uso interno das ferramentas que disponibilizarão as especificações. A especificação ainda define uma transformação, *RelationsToCore*, para

traduzir transformações definidas com a linguagem *Relations* para transformações definidas com a linguagem Core. Uma analogia citada na especificação (OMG, 2008a) é que a linguagem *Core* é similar ao *bytecode* da arquitetura Java, a semântica da linguagem *Core* é similar à especificação de comportamento da *Java Virtual Machine*, enquanto a linguagem *Relations* atua como a linguagem Java e a transformação *RelationsToCore* como o compilador *Java*. A Figura 2.3 ilustra o relacionamento entre estas linguagens.

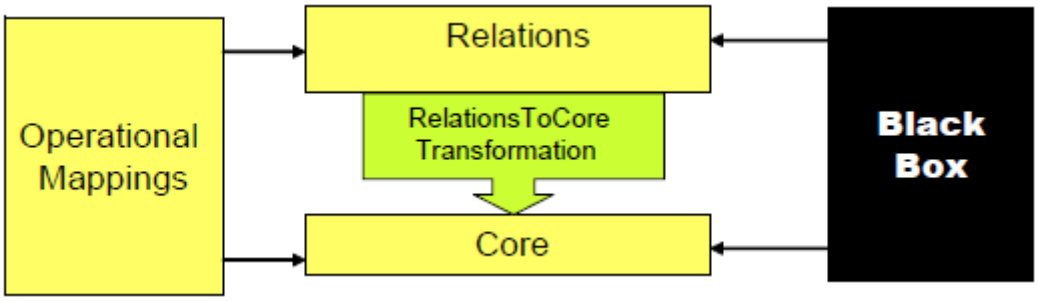

Figura 2.3 – Relacionamentos entre meta-modelos MOF QVT.
Fonte: OMG (2008a).

Uma transformação de modelos pode ser representada através do mapeamento de modelos de origem e destino baseando-se em regras pré-definidas de transformações sobre elementos dos meta-modelos. Isto é conceitualmente similar à compilação de código fonte (MRAIDHA, 2006). A Figura 2.4 descreve visualmente isso.

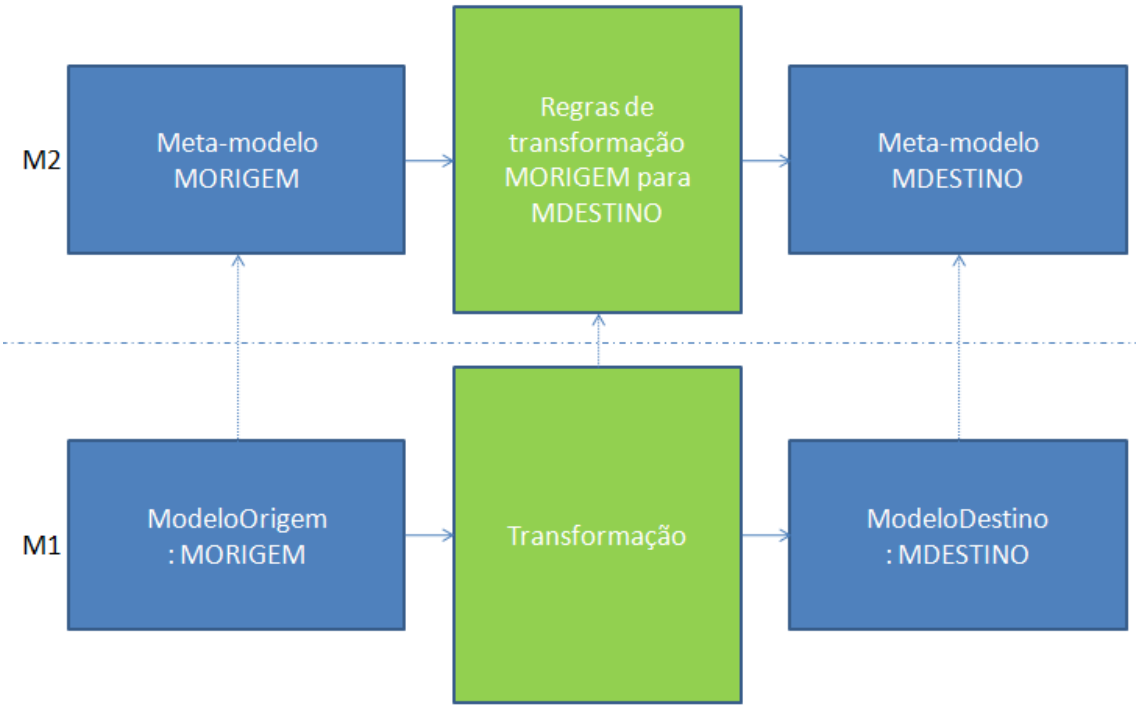

Figura 2.4 – Definição de transformações.
Fonte: adaptada de Mraidha (2006).

2.8.4.1. Entrelaçamento de modelos

Um caso especial de transformação de modelo para modelo, pouco explorado na literatura, é aquele onde existe a necessidade de entrelaçamento de modelos (*model weaving*). Um trabalho relevante na área é Marcos et al. (2005), que propõe uma abordagem para execução deste tipo especial de transformação, e também uma ferramenta.

A necessidade do entrelaçamento de modelos nasce naturalmente da MDA, pois ele sugere que modelos independentes de plataforma (PIM) devem ser entrelaçados com modelos de definição de plataforma (PDM) para a geração de um modelo específico de plataforma (PSM).

Marcos et al. (2005) define como um processo de entrelaçamento de modelos aquele que recebe dois modelos (m1) mais um meta-modelo de entrelaçamento (m2) e gera como resultado um modelo (m1). A partir do meta-modelo de entrelaçamento o processo define que elementos de cada modelo devem estar no modelo resultante, através de relacionamentos; e, em seguida, corrige todas as referências entre os elementos do modelo resultante.

Marcos et al. (2005) destaca quais pontos devem ser considerados na definição dos relacionamentos entre elementos dos dois modelos (m1) de entrada, sendo eles:

- O conjunto de relacionamentos não pode ser automaticamente gerado, pois ele é baseado em decisões de projeto ou em heurísticas;
- Deve ser possível salvar os relacionamentos para reutilizá-lo em outros contextos;
- Deve ser possível usar este conjunto de relacionamentos para ferramentas automáticas.

Sobre a motivação para a definição de operações de entrelaçamento, adicionais às transformações de modelo para modelo, Marcos et al. (2005) sugere os seguintes itens:

- Automação - uma transformação é uma operação automática enquanto que um entrelaçamento precisa de ajuda de heurísticas ou de guias para ajudar o usuário a executar a operação;
- Variabilidade - uma transformação é definida por um meta-modelo fixo (o meta-modelo da linguagem de transformação). Assim, enquanto não houver um meta-modelo padrão para entrelaçamento, cada aplicação precisa de um novo meta-modelo para entrelaçamento.

2.8.5. MOF M2T

Uma vez que existem os modelos PSM, eles precisam ser eventualmente transformados em artefatos de software, como código ou descritores de instalação. Para isso o OMG definiu a especificação MOF M2T (OMG, 2008b).

MOF M2T é uma especificação que define como traduzir um modelo para vários artefatos texto, como código, descritores de instalação, relatórios, documentos, etc... Uma maneira comum para implementar a transformação de modelos em uma representação textual é usar uma abordagem baseada em *templates*, onde o texto a ser gerado a partir dos modelos é especificado como um conjunto de *templates* parametrizados como elementos do modelo (OMG, 2008b). Assim como a especificação MOF QVT, a especificação MOF M2T reutiliza a linguagem OCL.

Segundo Mraidha (2006) existem duas abordagens para a geração de código a partir de modelos:

1. Mapeamento Implícito – onde as regras para geração do código estão dentro do transformador, ou seja, seguindo as especificações OMG tais regras são definidas usando-se MOF M2T;
2. Mapeamento Explícito – onde um modelo intermediário, contendo apenas os conceitos disponibilizados pela plataforma alvo, é introduzido; tal modelo na abordagem MDA é o PSM. Nesta abordagem, a especificação MOF QVT é usada uma ou mais vezes para gerar um PSM a partir do PIM. Então o PSM resultante é transformado em código, ou outros artefatos, usando-se a especificação MOF M2T.

Mraidha (2006) ainda apresenta os prós e contras de cada abordagem, os quais são apresentados na Tabela 2.1, adaptando-se às especificações propostas pelo OMG.

Tabela 2.1 – Prós e contras dos tipos de mapeamentos

Tipo de mapeamento	Prós	Contras
Implícito	Necessita somente da especificação MOF M2T	Transformador complexo, difícil de ser mantido Dificuldade para inserir novos mapeamentos
Explícito	Separação de responsabilidades, MOF QVT para gerar PSM e MOF M2T para gerar código Transformador simples	Necessita de dois tipos de transformações, m2m e m2t

Fonte: adaptada de MRAIDHA (2006)

2.8.6. Linguagens de ação

A declaração de uma classe em um diagrama de classes UML define a estrutura de parte do sistema, mas não define o comportamento desta parte. Na UML uma ação é a unidade fundamental para especificação de comportamento (OMG, 2009a). As ações executam algo, como criar ou destruir objetos.

A partir da versão 2.0 da UML uma semântica precisa para ações foi definida (MELLOR e BALCER, 2002), mais precisamente uma sintaxe abstrata para ações e atividades. Entretanto esta sintaxe abstrata não provê uma sintaxe concreta para uma linguagem de ação. Na UML a única sintaxe concreta definida é a gráfica, ou seja, os diagramas de comportamento, no caso de

ações mais notadamente diagramas de máquinas de estado e diagramas de atividades.

Segundo Mellor e Balcer (2002) não existe uma sintaxe concreta padrão para ações. Entretanto dado que a semântica (sintaxe abstrata) é independente da sintaxe concreta, usuários podem desenvolver sua sintaxe favorita ou linguagem. Mais, a definição de uma sintaxe padrão é um processo lento. Assim, a definição da semântica primeiramente permite que várias sintaxes concretas sejam desenvolvidas sobre uma mesma base. Mellor e Balcer (2002) deixam claro que a sintaxe concreta não é importante, e cita algumas linguagens concretas para ações, sendo elas: *BridgePoint Object Action Language*, *Small* (*Shlaer-Mellor Action Language*) e *Tall* (*That Action Language*).

2.8.7. fUML

fUML (*Semantics of a Foundational Subset for Executable UML Models*) é um subconjunto da UML formando uma linguagem computacional completa para modelos UML executáveis (OMG, 2009c).

A especificação fUML declara que seu propósito fundamental é servir como uma intermediária entre um subconjunto da UML usada para modelagem, e plataformas computacionais selecionadas para execução do modelo (OMG, 2009c). A Figura 2.5 exibe graficamente o propósito fundamental da fUML.

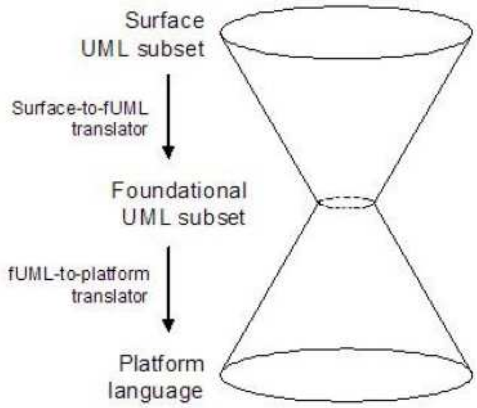

Figura 2.5 – Propósito fUML.
Fonte: OMG (2009c).

Para cumprir seu principal objetivo que é, servir como uma linguagem com nível de abstração mais alto, permitindo que modelos fUML sejam traduzidos para linguagens de computadores específicas para determinadas plataformas, foram considerados três critérios: tamanho reduzido, facilidade para tradução e funcionalidade para ações (OMG, 2009c).

Ela é um dos pilares para a MDA, pois MDA depende da noção de um modelo independente de plataforma (PIM), ou seja, um modelo que não esteja baseado em uma tecnologia de implementação (MELLOR e BALCER, 2002) (INCOSE, 2008).

fUML é o próximo nível de abstração a ser explorado na Engenharia de Software, abstraindo tanto a abordagem para implementação, quanto uma linguagem específica para programação, assim como decisões sobre a organização do software. Um modelo UML executável não especifica distribuição; ele não especifica o número e a alocação das tarefas (*tasks*). Assim, um sistema modelado com fUML pode ser instalado em vários ambientes de software sem alterações (MELLOR e BALCER, 2002).

Como os modelos fUML são completamente executáveis, fUML é uma forma para descrever um sistema utilizando modelos, inclusive estes modelos podem ser funcionalmente testados.

Um modelo executável tem basicamente três diagramas UML segundo (MELLOR e BALCER, 2002):

- Diagramas de classe, para modelar dados, descritos como classes, atributos, associações e restrições;
- Diagramas de transição de estados, para modelar controles, descritos como estados, eventos, transições e procedimentos;
- Diagramas de atividades, para modelar algoritmos, descritos como ações.

É importante destacar que segundo (MELLOR e BALCER, 2002) um diagrama de classes não deve conter operações, já que elas estão definidas no diagrama de transição de estados e podem ter comportamentos distintos para cada estado da classe.

2.8.7.1. fUML e seus blocos elementares

Antes de prosseguir com a revisão bibliográfica é preciso explorar o subconjunto da UML pertencente à fUML, apresentando de maneira breve os blocos elementares disponíveis na fUML para o projeto de um sistema.

As Figuras 2.6 e 2.7 apresentam os elementos fundamentais disponíveis na fUML para a definição de algoritmos. Os elementos destacados em azul são aqueles que foram usados no presente trabalho, e os comentários indicam os pacotes declarados na fUML.

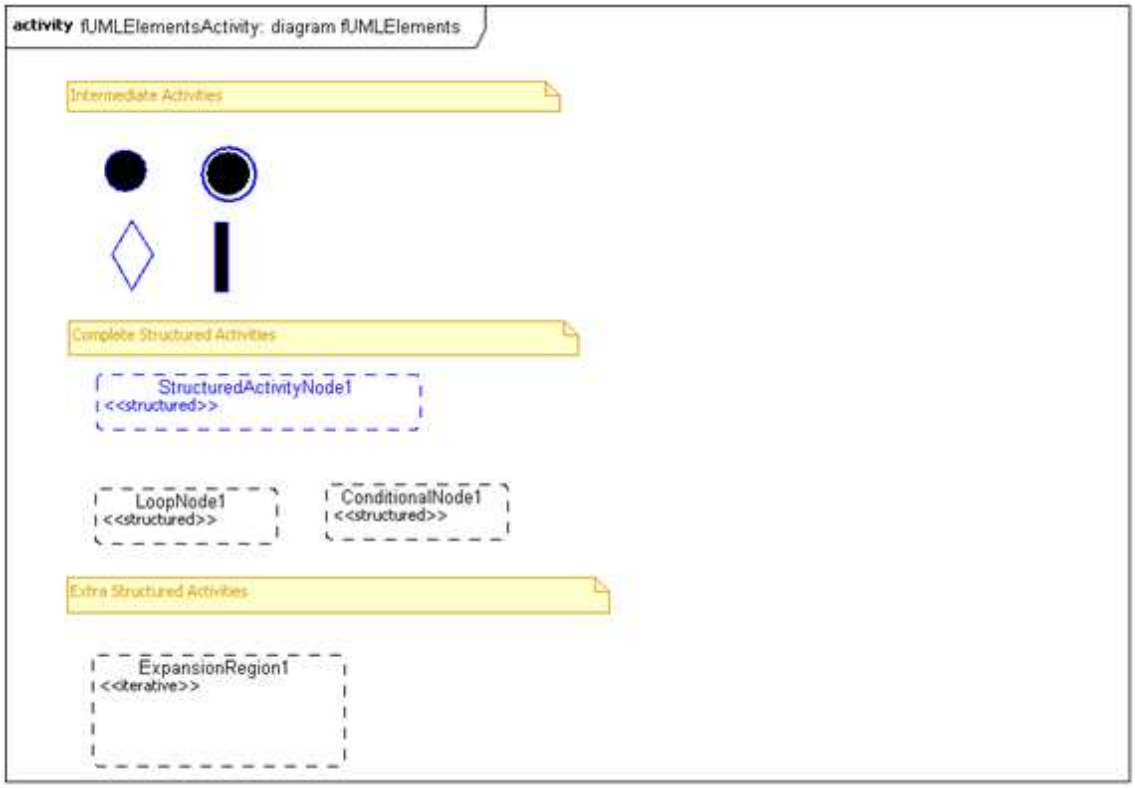

Figura 2.6 – Nós de controle disponíveis na fUML.
Fonte: OMG (2009c).

Importante é frisar que estes poucos blocos formam uma linguagem de programação de alto nível permitindo a definição de lógicas computacionais independente da plataforma de implementação. Por exemplo, existe na literatura (MELLOR e BALCER, 2002) registro de que um mesmo modelo PIM pode ser traduzido para linguagem C (uma linguagem estruturada) e também para C++ (uma linguagem orientada a objetos).

Diferentemente de linguagens de programação convencionais, não existe o conceito de uma linha principal de execução, a fUML é executada através de inúmeras máquinas de estados, todas elas sendo executadas de forma concorrente com chamadas assíncronas através de sinais, e síncronas através de chamadas a operações.

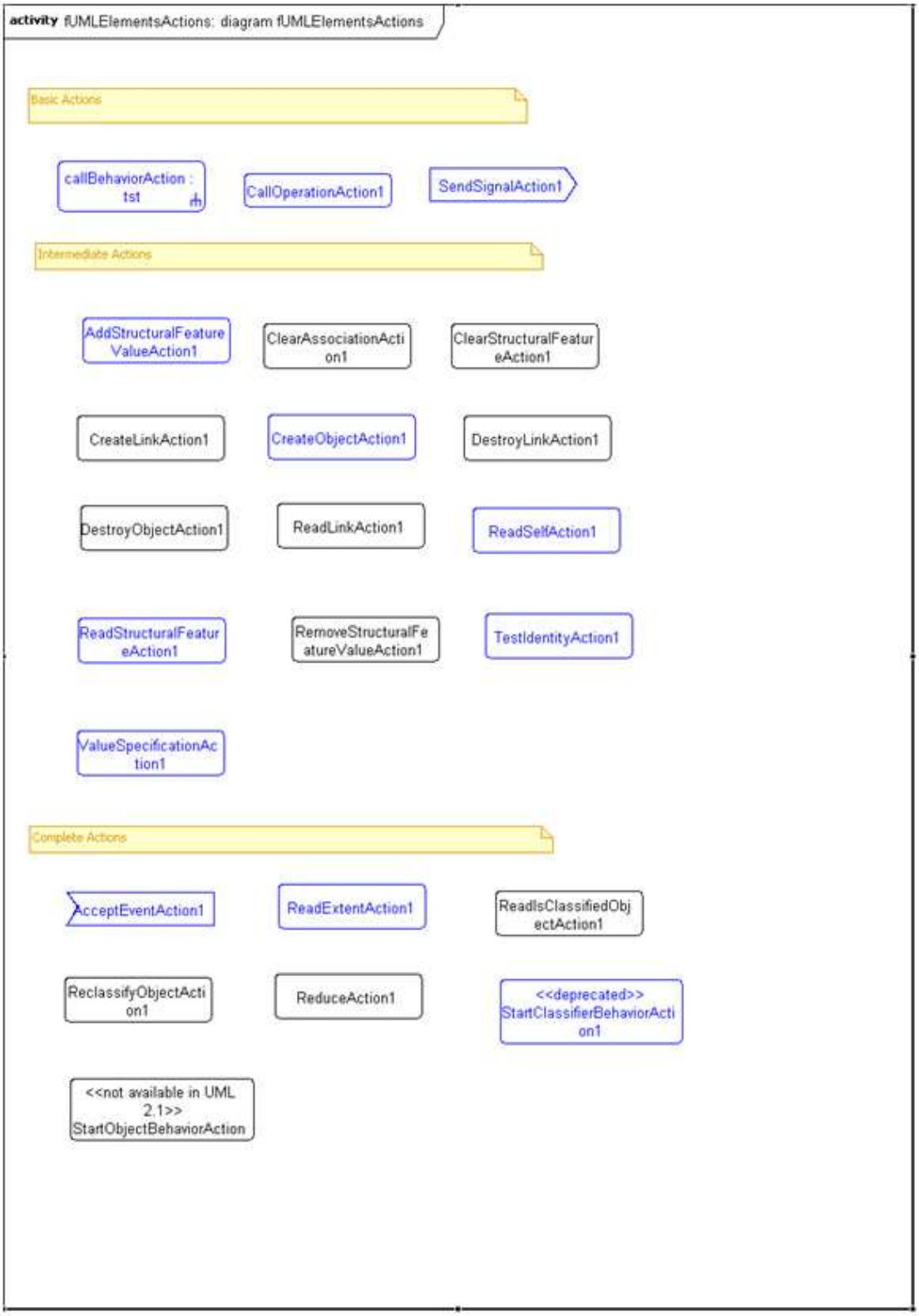

Figura 2.7 – Ações disponíveis na fUML.

Fonte: adaptada de OMG (2009c).

Para compreender como um algoritmo computacional é definido utilizando-se fUML é preciso entender, mesmo que superficialmente, os blocos elementares disponibilizados por ela. Tais blocos são uma generalização de instruções disponíveis em linguagens de programação como Java ou C++.

Os blocos elementares de controle que precisam ser apresentados são:

- *Merge e Decision* – usados para decisões baseadas em guardas (*guard*) e também para laços;
- *Fork e Join* – usados para separar ou juntar fluxos de controle concorrentes, usados também para reuso de objetos;
- *StructuredActivity* – usados para isolar logicamente partes de atividades.

Os blocos elementares de ações da fUML que precisam ser apresentados são:

- *CreateObjectAction e DestroyObjectAction* – usados para criar ou destruir objetos;
- *ReadStructuralFeatureAction e AddStructuralFeatureAction* – usados para adicionar ou ler propriedades de objetos;
- *ReadSelfAction* – usado para obter-se uma referência ao classificador corrente;
- *TestIdentityAction* – usado para testar se dois objetos são iguais;
- *ValueSpecificationAction* – usado para definir-se constantes;
- *SendSignalAction e AcceptEventAction* – usados para enviar ou receber sinais;
- *CallBehaviorAction e CallOperationAction* – usados para disparar comportamentos através de atividades ou operações; Um ponto a ser destacado é que fUML prevê polimorfismo nos disparos de operações;
- *ReadExtent* – usado para localizar instâncias de um classificador e de seus filhos.
- *StartClassifierBehaviorAction* – usada para inicializar um *ClassifierBehavior* de um objeto ativo. A fUML (OMG, 2009c) declara

que esta ação deve ser encarada como depreciada, sendo substituída pela *StartObjectBehaviorAction*.

Os blocos elementares de objetos da fUML que precisam ser apresentados são:

- *InputPin e OutputPin* – usados para receber ou retornar objetos.

Está fora do escopo deste trabalho explicar detalhes sobre cada nó de controle, objeto ou ação. Para aprofundamento no tema a literatura é vasta, destacando-se a própria especificação UML (OMG, 2009b).

2.8.8. MARTE

Como UML é uma linguagem de modelagem de propósito geral, ela não possui construções e abstrações para representar conceitos específicos de sistemas embarcados de tempo real. A primeira tentativa para tratar esta deficiência foi o *UML Profile for Schedulability, Performance and Time Specification* (SPT) (OMG, 2009a). SPT provia conceitos para permitir a análise de escalonabilidade assim como análise de performance, além de possuir meios para modelar o tempo e mecanismos relacionados ao tempo. Entretanto, o mercado localizou uma grande quantidade de problemas ao tentar utilizar o SPT. Então um novo UML *Profile* para sistemas embarcados de tempo real foi desenvolvido (OMG, 2009a).

MARTE (*UML Profile for Modeling and Analysis of Real-Time Embedded Systems*) (OMG, 2009a) é o novo UML *Profile* definido pelo OMG, ele trata: projeto de aspectos de software e hardware em sistemas embarcados; alocação de elementos; capacidade ampliada para análise de escalonabilidade e performance; especificação de características de sistemas embarcados, como capacidade de memória e consumo de energia; suporte a arquiteturas

baseadas em componentes; outros paradigmas computacionais, como assíncrono e síncrono (WEHMEISTER, 2009).

De fato, MARTE foi projetado para tratar dois itens: modelar características de sistemas embarcados de tempo real e suportar a análise de propriedades do sistema (WEHMEISTER, 2009).

A especificação MARTE teve uma enorme contribuição da *Architectural Analysis and Design Language* (AADL). Inclusive consta da especificação (OMG, 2009a) um anexo sobre equivalências entre conceitos MARTE e AADL.

AADL é uma linguagem de descrição arquitetural (*architectural description language*, ADL) que fornece uma sintaxe e uma semântica para descrever uma arquitetura de software. Uma ADL deve fornecer ao projetista a habilidade de decompor componentes arquiteturais, compor componentes individuais em blocos arquiteturais maiores e representar as interfaces (mecanismos de conexão) entre componentes (PRESSMAN, 2006).

Mais especificamente, a AADL é uma linguagem de modelagem que suporta análise prévia e repetida da arquitetura de um sistema com relação a propriedades não funcionais (NFPs) críticas através de uma notação extensível, um quadro de trabalho (*framework*) para ferramentas, e uma semântica precisa. Ela é especialmente efetiva na análise baseada em modelos e na especificação de sistemas embarcados de tempo real (FEILER et al., 2006).

Segundo Pressman (2006) UML pode ser considerara uma ADL, pois inclui muitos artefatos necessários para descrições arquiteturais, mas não é tão completa quanto a AADL. Entretanto o UML *Profile* MARTE visa completar a UML trazendo todos os elementos necessários para a modelagem de arquitetura para sistemas embarcados de tempo real.

É importante destacar que segundo Feiler et al. (2006) um dos grandes fatores que geraram a necessidade do MARTE é a falta de expressividade da UML nos modelos PSM, ou seja, o uso do MARTE está mais direcionado a modelos PSM.

2.9. Critérios para estruturação de subsistemas

Um ponto importante no presente trabalho é usar a conhecida abordagem “dividir para conquistar”. De acordo com esta abordagem, um sistema deve ser dividido em subsistemas menores para facilitar sua compreensão e modelagem.

Sobre esta questão, Pressman (2006) apresenta a Figura 2.8 destacando que é preciso buscar a região de custo mínimo. Em tal região são evitados desvios para a submodularização ou para a supermodulariazação.

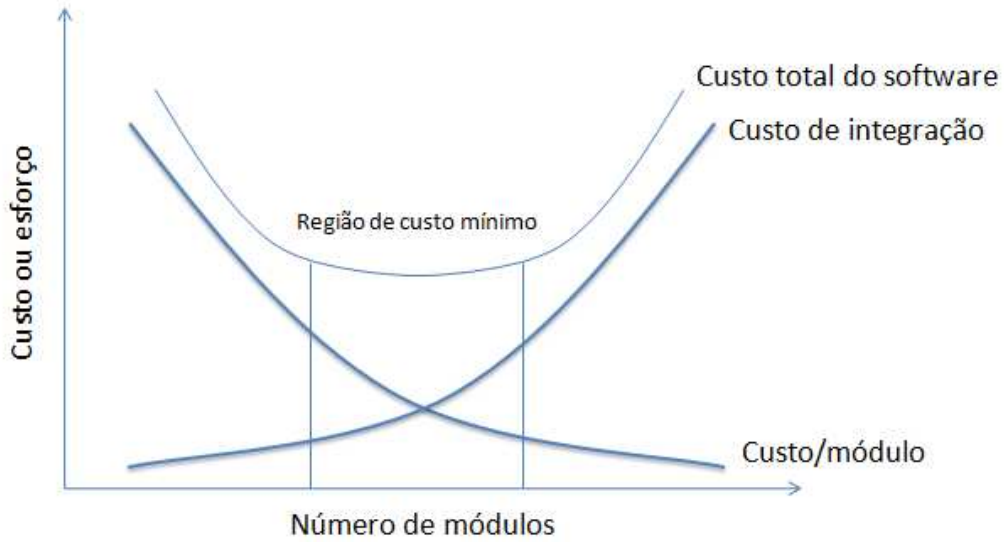

Figura 2.8 – Modularidade e custo de software.
Fonte: adaptada de Pressman (2006).

Outro ponto relevante para a estruturação de subsistemas são os variados tipos de coesão. Pressman (2006) define sete tipos de coesão, dentre eles

quatro se destacam, sendo eles (listados em ordem decrescente de nível de coesão):

- Coesão funcional – quando o critério de agrupamento é a afinidade funcional;
- Coesão comunicacional – quando o critério de agrupamento é a afinidade para manipulação de dados;
- Coesão seqüencial – quando componentes ou operações são agrupados para permitir que o primeiro forneça entrada para o próximo e assim por diante; Gomaa (2000) estabelece um critério para o que ele chama de agrupamento de tarefas seqüenciais (*Task Sequential Clustering*);
- Coesão temporal – quando o aspecto temporal é o critério considerado para agrupamento, por exemplo, a tarefa um e a tarefa dois sempre são executadas com mesmo período e prazo, conseqüentemente elas podem ser agrupadas; Gomaa (2000) estabelece um critério para o que ele chama de agrupamento de tarefas baseado no tempo (*Task Temporal Clustering*).

Gomaa (2000) define alguns tipos de subsistemas, sendo os principais: controle (*control*), coordenador (*coordinator*), recuperador de dados (*data collection*), analisador de dados (*data analysis*), servidor (*server*) e interface de usuário (*user interface*).

Além desses pontos pelo menos três princípios básicos de projeto (Pressman, 2006) precisam ser resgatados, sendo eles:

- Princípio de Equivalência entre Liberação e Reuso (*Release Reuse Equivalency Principle, REP*) – Ele determina que a granularidade do reuso é a granularidade da versão;
- Princípio do Fecho Comum (*Common Closure Pinciple, CCP*) - Ele determina que classes que se modificam juntas devem ficar juntas;

- Princípio Comum de Reuso (*Common Reuse Principle, CRP*) – Ele determina que elementos que não são reusados juntos não devem ser agrupados juntos.

2.10. PMM

A PMM (Plataforma Multi-Missão) foi concebida como sendo uma plataforma versátil, podendo operar em órbitas de baixa inclinação e polares, com apontamento terrestre e inercial, de uso em várias missões de satélites do Plano Nacional de Atividades Espaciais (PNAE) (AEB, 2005). As missões que irão utilizar a PMM são: Satélite de Sensoriamento Remoto (SSR-1 e 2) e satélite de Coleta de Dados (SCD-3) (MOREIRA, 2006).

Com o desenvolvimento da PMM, o INPE (Instituto Nacional de Pesquisas Espaciais) irá adquirir a tecnologia de satélites estabilizados em três eixos com apontamento fino. Esse é um passo fundamental rumo à autonomia brasileira no campo da tecnologia de satélites (MOREIRA, 2006).

Para cumprir os requisitos da missão, o subsistema de controle de atitude e órbita deve apresentar em seu computador de bordo um sistema operacional com características de tempo real. Estas características de tempo real supõem a previsibilidade das tarefas executadas no computador de bordo, garantindo então um alto nível de confiabilidade do software de controle (MOREIRA, 2006).

2.11. Trabalhos correlatos

MDA advoga que, com a existência dos modelos PIM e de um processo automatizado para transformação dos mesmos em modelos PSM e depois em código, vários benefícios discutidos anteriormente serão realizados.

Nesta linha o trabalho de Wehrmeister (2009) propõe um método para analisar e desenvolver modelos, um conjunto de quadros de trabalhos e uma ferramenta para transformar modelos em código. Nele um modelo UML (PIM) é traduzido em outro modelo PIM, chamado de DERCS – *Distributed Embedded Real-Time Compact Specification*, composto por parte da UML e por parte do MARTE, usando uma transformação customizada que recebe regras através de um arquivo XML. Em seguida, outra transformação parte do PIM (DERCS) para a geração de código baseando-se em um script definido em um arquivo XML. Este trabalho se preocupa tanto com a modelagem de estrutura quanto de comportamento, para modelar comportamento à opção selecionada é usar diagramas de seqüência com uma linguagem de ação customizada.

Já Feiler et al. (2007) propõe o uso de um *profile* customizado da UML para a definição do PIM, focado essencialmente nas estruturas de software sem se preocupar com o ambiente de execução. A partir do qual um modelo AADL é criado descrevendo o PSM, focado no ambiente de execução.

Para a análise de escalonabilidade, Maes (2007) propõe uma transformação usando uma ferramenta chamada ATL (ECLIPSE FOUNDATION, 2010a) para transformar modelos UML, anotados com o *profile* MARTE, em arquivos XML no formato esperado pela ferramenta Cheddar (LISYC, 2010).

Já Moreira (2006) usa a ferramenta MATRIXx para modelar o subsistema de controle de atitude e órbita da PMM, desde a definição dos algoritmos de controle até a geração de código e avaliação em um ambiente simulado de tempo real.

Silva (2005) apresenta um estudo de caso focado essencialmente nas especificações OMG para transformação de estrutura de modelos. Neste trabalho não existe a preocupação com a geração de comportamento da

aplicação. Ele parte de um modelo PIM, modelando apenas a estrutura, e gera o código para o mesmo.

Wang (2005) apresenta uma avaliação do status das especificações OMG e ferramentas a serem utilizadas na transformação de modelos.

Marcos et al. (2005) está focado no entrelaçamento de modelos, apresentando uma abordagem para definição dos meta-modelos de entrelaçamento assim como uma ferramenta visual para isso.

Não foram localizados trabalhos que explorassem a fUML. O principal motivo atribuído a esta ausência de trabalhos é que a versão beta foi publicada em Novembro/2009.

Avaliando estas pesquisas de forma abrangente, é possível notar que poucos trabalhos avaliam como a Engenharia Conduzida por Modelos, ou mais especificamente, a MDA com suas especificações, pode contribuir para aumentar a maturidade da Engenharia de Software. Corroborando esta afirmação, um fato importante é que nenhum dos trabalhos avaliados, quando avaliam todo o ciclo de vida (Silva, 2005) (Wehrmeister, 2009), usam explicitamente o PSM. Sem usar o PSM é muito difícil fazer qualquer tipo de análise de escalonabilidade.

Sobre as especificações, nenhum trabalho explora a MOF QVT ou a MOF M2T.

Várias lacunas existem, sendo uma fundamental: como tratar propriedades não funcionais no contexto da MDA? Devem ser modeladas no PIM ou no PSM?

O próximo capítulo apresenta a proposta avaliada no presente trabalho, a qual foi definida com base na revisão bibliográfica apresentada neste capítulo.

3 PROPOSTA

Este capítulo apresenta a proposta utilizada no presente trabalho para a modelagem de sistemas de tempo real embarcados. Primeiramente são apresentadas as considerações que serviram de base para a concepção da proposta, em seguida uma visão geral da arquitetura. Então cada parte da arquitetura proposta é detalhada e finalmente as ferramentas selecionadas para o presente trabalho são apresentadas.

3.1. Considerações para concepção da proposta

Antes de apresentar a proposta, é preciso discorrer sobre as considerações avaliadas durante a concepção da proposta.

MDA é uma das abordagens para utilização da MDE, entretanto tem se destacado como uma alternativa promissora. Um exemplo disso é o volume de trabalhos acadêmicos focados nesta abordagem (SILVA, 2005) (WEHRMEISTER, 2009).

MDA é um elemento chave na promoção da produtividade no desenvolvimento de software, assim como a portabilidade e a manutenabilidade de sistemas de software (FLINT e BOUGHTHON, 2003). Como explorado no capítulo anterior, a MDA sugere o uso de um modelo PIM, que é automaticamente transformado em um modelo PSM, que por sua vez é automaticamente transformado em código. Alguns trabalhos, como Silva (2005) destacam estes modelos. Entretanto, ainda seguindo a MDA um terceiro modelo faz-se necessário, é o PDM (*Platform Description Model*) ou simplesmente PM (*Platform Model*). O PDM prove um conjunto de conceitos técnicos, como classes ou serviços disponibilizados por uma plataforma.

A Figura 2.9 mostra o relacionamento entre estes modelos e transformações, completando a Figura 2.2.

É importante explorar a necessidade da existência de um modelo PSM. Já foi apresentado que Mraidha (2006) define duas alternativas para a geração de código a partir de modelos: mapeamento implícito e explícito. As vantagens relevantes para o presente trabalho do mapeamento explícito são (Mraidha, 2006):

- Separação de responsabilidades, pois existe uma transformação para traduzir um PIM para um PSM considerando um dado PDM;
- Simplificação da transformação que gera código;
- Permite a análise de propriedades do modelo.

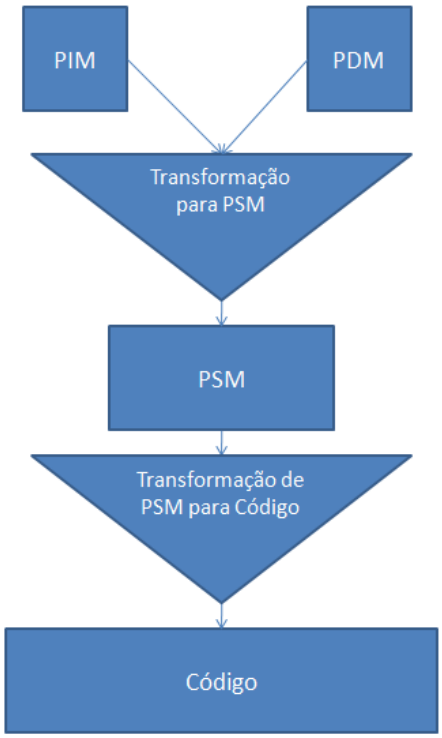

Figura 2.9 – Abordagem MDA com PDM.
Fonte: adaptada de Marcos et al. (2005).

Considerando as vantagens do mapeamento explícito, o presente trabalho utiliza esta abordagem, ou seja, considera importante a existência de um modelo PSM. Para corroborar esta decisão, outro fator importante gera a

necessidade de um PSM é a falta de expressividade para manipular modelos da especificação MOF M2T (OMG, 2008b), usada para transformar modelos em código, em contraste com a alta expressividade da especificação MOF QVT (OMG, 2008a) aplicada nas transformações de modelo para modelo.

Sobre a especificação MOF QVT, conforme explorado nos capítulos anteriores ela é dividida em três partes: *core*, *relations* e *operational*. Segundo Dvorak (2008) o QVT *Operational* é mais adequado para transformações que construam modelos com uma estrutura complexa. Como o presente trabalho lida com transformações para modelos com estrutura complexa, como tradução de atividades UML, ele utiliza para todas as transformações de modelos para modelos o QVT *Operational*. Outro ponto importante, é que, até o momento da concepção da arquitetura do presente trabalho, não existiam ferramentas maduras disponibilizando o QVT *Relations*.

A abordagem definida até aqui é suficiente para o desenvolvimento de software em geral, entretanto para o desenvolvimento de software de tempo real embarcado é preciso complementar esta macro arquitetura com pelo menos três definições:

1. Como integrar os modelos com ferramentas usadas na engenharia de controle, como o Matlab e o MATRIXx?
2. Onde definir características da plataforma de hardware?
3. Onde definir as propriedades não funcionais do software, como os requisitos de tempo?

Sobre a integração com ferramentas usadas na Engenharia de Controle, segundo Vanderperren e Dehaene (2006) existem duas abordagens possíveis:

- Co-simulação, ou seja, onde as simulações UML e Matlab Simulink se comunicam via uma ferramenta intermediária.
- Integração baseada em uma linguagem comum de execução, ou seja, gerar código para o modelo UML e uní-lo ao código gerado pelo Matlab.

Ainda sobre a integração com ferramentas usadas na Engenharia de Controle, é preciso avaliar as iniciativas do OMG para a definição de restrições. A principal iniciativa é o diagrama paramétrico definido na SysML, que descreve restrições especificadas por equações com variáveis. Segundo Espinoza (2008) tais equações podem ser descritas usando-se uma linguagem de terceiros, como MathML ou Modelica. Outra iniciativa do OMG é a linguagem VSL (*Value Specification Language*) definida no MARTE (OMG, 2009a), ela define uma sintaxe para formular expressões algébricas e de tempo. Espinoza (2008) sugere o uso combinado de diagramas paramétricos da SysML e da VSL: enquanto os diagramas paramétricos disponibilizam um formalismo amigável para definir modelos não causais, VSL prove a sintaxe para expressões de restrição.

O presente trabalho não foca na modelagem das restrições, e quando elas precisam ser modeladas opta-se pela definição de tais restrições usando-se UML. É preciso destacar que o objetivo não é substituir as ferramentas de Engenharia de Controle por UML, mas sim mostrar que a modelagem de restrições em UML é possível. Um ponto a ser destacado é que esta abordagem não depende da existência de uma linguagem comum de execução, permitindo que se o código for gerado em Java que ele não envolva chamadas a código C++ ou Ada (linguagens usuais para geração de código em ferramentas da engenharia de controle), garantindo a portabilidade do código a ser gerado.

Sobre onde definir características da plataforma de hardware, o local adequado é o PDM. Entretanto para permitir que a plataforma de hardware varie livremente das características da plataforma de software o PDM deve ser dividido em pelo menos dois modelos:

1. PDM Hardware – que define as características da plataforma de hardware alvo;

2. PDM Software – que define os principais conceitos e serviços da plataforma de software.

Esta divisão gera a necessidade de uma transformação capaz de fundir estes dois modelos em um único modelo (PDM).

Sobre onde definir propriedades não funcionais do software, o local mais adequado é no modelo PIM, registrando-se desde o início tais propriedades e as utilizando em todos os modelos. Por exemplo, a indicação de com que periodicidade um sensor deve ser lido deve ser documentada no PIM.

Entretanto, propriedades derivadas das características do código gerado em execução em determinada plataforma, só podem ser informadas no PSM. Um exemplo é o tempo de ativação de um determinado método usando-se a linguagem escolhida no hardware definido.

Outro ponto a ser esmiuçado, é o quanto um PIM é realmente independente de plataforma, e em que aspectos ele é independente?

Segundo o INCOSE (2008) um PIM exibe um grau específico de independência de plataforma e é mais adequado para uso com um número diferente de plataformas do mesmo tipo. Tal afirmação determina que um PIM é independente até um determinado grau, por exemplo, para se construir o modelo PIM para o sistema de controle de atitude da PMM a arquitetura geral do sistema é uma informação de entrada, quantos sensores existirão, com que freqüência deve-se lê-los, etc... A independência é exibida mais fortemente em dois aspectos:

1. Quanto ao ambiente de execução, hardware, sistema operacional, linguagem de programação, etc...

2. Quanto às alternativas para implementação, que algoritmos usar, como distribuí-los, como implementar a comunicação entre os módulos, etc...

O item 2 define a capacidade de comparação entre soluções que gerem o mesmo resultado, permitindo-se a exploração do espaço de projeto e a otimização da solução final. Este item ainda contribui fortemente no tópico tolerância a falhas. Neste momento cabe destacar a técnica chamada “Redundância através de Projeto Diverso” (*Diverse Design Redundancy*), que consiste no desenvolvimento de dois ou mais componentes de desenho diferente para prover o mesmo serviço (SOUZA e CARVALHO, 2005), o que permite proteção contra falhas causadas por deficiências no projeto. Já que é possível transformar um mesmo modelo PIM em mais de um PSM que deverá exibir o mesmo resultado final.

O último ponto considerado na definição da proposta é sobre qual plataforma de software deveria ser usada para avaliação. Java tem se mostrado extremamente eficaz, com aceitação cada vez maior no meio acadêmico e comercial. Além disso, para sistemas de tempo real a extensão Java Real Time tem chamado cada vez mais a atenção de empresas, assim como de centros de pesquisa. Não é escopo, do presente trabalho, apresentar as vantagens e desvantagens da plataforma Java, e, para fins da avaliação de aplicabilidade, a extensão Java Real Time não foi selecionada. Mas cabe aqui destacar pelo menos um ponto: portabilidade, já que a definição de uma plataforma (PDM) em Java aumenta as possibilidades de reutilização, dado que variações no hardware ou no sistema operacional não demandam alterações na PDM. Uma discussão mais aprofundada sobre o uso de Java ou não em software embarcado pode ser encontrada em Pasetti (2002).

3.2. Visão geral da arquitetura

O presente trabalho assume que, em um processo típico de Engenharia de Sistemas, a definição da arquitetura do sistema, a elucidação dos requisitos de sistema e de subsistemas assim como a modelagem de casos de uso do sistema e subsistemas estão prontos. Estas informações são usadas como entrada para a arquitetura do presente trabalho, a partir das quais duas distintas tarefas são inicialmente executadas:

- Modelagem de negócio, executada através da modelagem de PIMs;
- Definição de uma plataforma de destino, envolvendo modelos PDM e transformações de modelo para modelo e de modelo para código.

Para definição do modelo PIM a especificação fUML é utilizada. No modelo PIM propriedades não funcionais serão documentadas, usando-se o UML *profile* MARTE. Como este modelo PIM é construído usando-se fUML, uma ferramenta capaz de interpretar fUML é capaz de permitir o teste funcional de tais modelos.

Para definição do modelo PDM, é usada UML aplicando-se o UML *profile* MARTE para definição de características de hardware e de conceitos da plataforma Java.

A partir dos modelos PIM e PDM uma transformação, usando a especificação MOF QVTO, é responsável por gerar o PSM. O PSM é um modelo UML automaticamente instrumentado com o UML *profile* MARTE.

A partir deste PSM duas novas transformações são realizadas, usando a especificação MOF M2T, do PSM para código Java e também do PSM para um modelo AADL. A transformação do PSM para Java será focada em uma arquitetura e plataforma (KENNEDY CARTER, 2002), ou seja, para cada arquitetura e plataforma uma nova transformação deve ser definida e

desenvolvida. Já a transformação de PSM para AADL é genérica e poderá ser usada para qualquer modelo PSM, que respeite as regras definidas na transformação.

Uma plataforma do ponto de vista da MDA é formada por PDMs e também por transformações de modelo para modelo e de modelo para código.

O modelo AADL é muito importante na presente proposta, pois, a partir dele, a análise de escalonabilidade é realizada.

A arquitetura proposta minimiza a necessidade de intervenções em modelos intermediários ou no código, no caso de alterações ou correções os modelos a serem alterados ou revisados são aqueles que estão no topo da arquitetura, sendo eles PIM ou plataforma (PDM + transformações). O(s) modelo(s) intermediário(s), PSM, e até mesmo o código final, são obtidos utilizando-se a abordagem de desenvolvimento de software baseada em transformações, ou seja, não devem ser alterados.

A Figura 3.1 descreve toda a arquitetura proposta. No funil à esquerda o elemento “Sistema Subsist n” representa os requisitos e casos de uso de sistema e de cada subsistema.

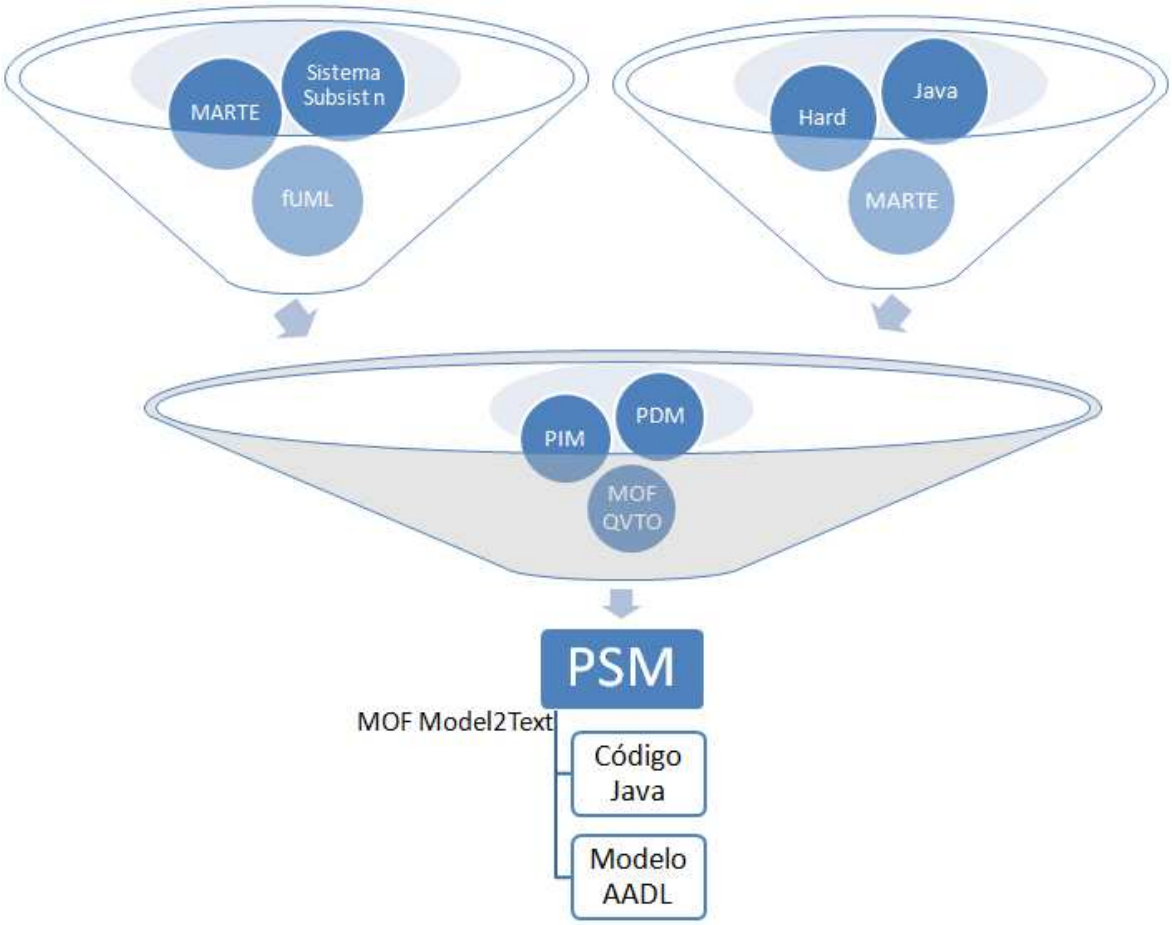

Figura 3.1 – Arquitetura proposta.

Em seguida cada parte da arquitetura é apresentada em mais detalhes.

3.3. Detalhamento da arquitetura

As próximas seções detalham a arquitetura explorando cada parte apresentada na visão geral. Primeiramente apresenta-se a abordagem para modelagem do PIM, representado na Figura 3.1 no canto superior esquerdo. Em seguida, o PDM é detalhado, representado na Figura 3.1 no canto superior direito. Finalmente os outros elementos - PSM, código (Java no presente trabalho) e AADL - são apresentados.

3.3.1. PIM

Conforme explorado nos capítulos anteriores, MDA depende da noção de um modelo independente de plataforma (PIM), ou seja, um modelo que não esteja baseado em uma tecnologia de implementação (MELLOR e BALCER, 2002) (INCOSE, 2008), sendo fUML uma alternativa para a definição deste modelo (OMG, 2009c).

A especificação fUML também a descreve como uma linguagem intermediária entre um subconjunto da UML e uma plataforma definida. Mas a pergunta é: O subconjunto da UML definido na fUML é o suficiente para modelar sistemas no nível de abstração dos modelos PIM? O presente trabalho considera que sim, seguindo também (MELLOR e BALCER, 2002) que advoga o uso apenas de um subconjunto da UML (similar à fUML, trabalho inicial que deu origem a RFP para fUML) para modelagem de sistemas no nível de abstração de modelos PIM.

A fUML define uma sintaxe abstrata (representação formal dos conceitos) limitando-se principalmente à sintaxe abstrata de classes e atividades, mas não define uma sintaxe concreta. Assim cada provedor de ferramenta pode definir uma sintaxe concreta, gráfica ou textual, para os conceitos definidos na fUML.

Não é escopo, do presente trabalho, propor uma sintaxe concreta para fUML. Assim, a sintaxe concreta definida na UML é reutilizada. Como o único tipo de comportamento definido por usuário suportado pela fUML é a atividade, cada comportamento no modelo PIM deve ser modelado como uma atividade (OMG, 2009c), e conseqüentemente, tal atividade é visualizada por um diagrama de atividades.

Mellor e Balcer (2002) frisam que desenvolver sistemas envolve o entendimento de diferentes partes e de como ligá-los para criar um todo coerente. Cada parte semanticamente autônoma é um domínio (*domain*), capaz de ser entendido e modelado usando-se UML (MELLOR e BALCER, 2002). O presente trabalho prefere chamar cada parte semanticamente autônoma de subsistema, para evitar confusões com o uso já estabelecido do termo “domínio” na Engenharia de Software, além de alinhar-se com a Engenharia de Sistemas.

Neste ponto é importante destacar que um dos objetivos do MDA é o reuso de conhecimento especializado (*expertise*), materializado em modelos, e não reuso de código. O reuso de código normalmente está associado à Engenharia de Software baseada em componentes (*Component-Based Software Engineering - CBSE*), que é definida por Pressman (2006) como um processo que enfatiza o projeto e a construção de sistemas baseados em computador usando “componentes” de software reutilizáveis.

Visando o reuso de conhecimento, através de modelos, é importante considerar o Princípio da Equivalência entre liberação e Reuso (Pressman, 2006). Este princípio determina que a granularidade de reuso é a granularidade da liberação, ou seja, cada subsistema autônomo deve ser definido em um modelo PIM independente. Isto permite, por exemplo, que um subsistema para leitura de sensores usado na PMM possa ser usado em outro satélite com características arquiteturais similares.

Sobre PIMs independentes para cada subsistema é preciso ainda responder a três perguntas fundamentais:

1. Como modelar os subsistemas de forma independente?
2. Como testar subsistemas?
3. Como entrelaçar n modelos de subsistemas em um único um sistema?

A resposta à primeira pergunta depende da introdução do conceito de terminadores (*terminators*), que conforme definido em Kennedy Carter (2003), representam abstrações de entidades externas expressas em um subsistema. Pois subsistemas são autônomos, entretanto eles se baseiam na existência de outros subsistemas (MELLOR e BALCER, 2002). No caso do subsistema de leitura de sensores da PMM, seu principal objetivo é prover informação para o subsistema de controle, e tal relacionamento precisa ser documentado de alguma forma, ou seja, a entidade controlador é modelada no subsistema de leitura de sensores da PMM como um terminador. Na presente proposta terminadores são modelados como classes abstratas, contendo apenas os detalhes necessários para uso no subsistema sendo modelado.

Como existem dois ou mais subsistemas autônomos com terminadores modelados, Mellor e Balcer (2002) definem o conceito de ponte (*bridge*) que é uma espécie de dependência entre subsistemas. O presente trabalho implementa o conceito de ponte através de uma transformação MOF QVTO responsável por entrelaçar dois PIMs independentes, substituindo terminadores (classes abstratas) por classes concretas. Isto responde à segunda pergunta. Ainda sobre o conceito de ponte, Mellor e Balcer (2002) distinguem dois tipos de pontes, as explicitas e as implícitas. As explicitas são aquelas onde um elemento de um subsistema faz referência direta a um elemento externo, causando um acoplamento entre dois subsistemas. Um exemplo concreto deste tipo de ponte é a herança. As pontes implícitas são aquelas onde não existe referência a um elemento externo, mas sim a terminadores que podem ser substituídos por qualquer elemento externo. Um exemplo concreto deste tipo de ponte é a orientação a aspectos, pois os aspectos podem ser entrelaçados a qualquer classe de forma declarativa.

Antes de responder à terceira pergunta é preciso considerar as alternativas para teste de um subsistema. É possível testá-lo de forma integrada, testando

o sistema e o subsistema simultaneamente. Entretanto, é desejável que um teste apenas do subsistema seja realizável permitindo a obtenção de resultados mais rápidos e com mais correção. Agora, para testar de forma isolada um subsistema que depende de um terminador é preciso simular este último. Se é preciso modelar um simulador, este deve ser modelado como outro PIM (subsistema de teste do subsistema inicial), permitindo que este seja reutilizado para testes subseqüentes. Mas se já existe um modelo para teste, porque não documentar as condições de teste usando-se fUML e executá-las novamente a cada alteração no modelo do subsistema? Assim, é possível usar o conceito de integração contínua aplicada, normalmente associada a código, também a modelos, ou seja, a partir de uma determinada condição de disparo o modelo do subsistema e o modelo de teste do subsistema são entrelaçados, executados usando-se uma ferramenta capaz de interpretar fUML; e, durante esta execução, as condições de teste são avaliadas automaticamente. Ainda sobre testes, os processos ágeis propuseram a construção de código de teste antes da construção do próprio código. Tal abordagem pode ser seguida também com modelos (MELLOR e BALCER, 2002). Mellor e Balcer (2002) destacam finalmente que a maior contribuição dos modelos executáveis é justamente antecipar a verificação de sistemas.

Especificamente sobre sistemas de tempo real restam ainda dois pontos a serem explorados:

1. Semântica do tempo - Como modelar tempo?
2. Como definir propriedades não funcionais (NFPs) em um PIM?

fUML não restringe explicitamente o modelo de execução quanto à semântica do tempo. Assim, é preciso definir este ponto para permitir a modelagem de um sistema de tempo real. O presente trabalho optou por modelar uma biblioteca, chamada *ClockLibrary,* usando a própria fUML. Ela supõe um relógio centralizado e ideal, mais detalhes sobre esta biblioteca serão apresentados no próximo capítulo. Vale a pena salientar que a semântica de tempo para o

sistema final, código gerado em Java no presente trabalho, é definida parte pela plataforma alvo e parte pela transformação dos modelos PIM para PSM e que pode ser diferente da definida no modelo PIM.

Sobre como definir propriedades não funcionais em um PIM, o UML *profile* MARTE prove um pacote especificamente para isso. Ele se chama *High-Level Application Modeling* (HLAM). Diferentemente de outros tipos de aplicações, um sistema de tempo real necessita de possibilidades adicionais para modelagem de características quantitativas como, por exemplo, período e prazo. O principal objetivo do HLAM é prover tais conceitos para uma modelagem de alto nível (OMG, 2009a). O HLAM define vários locais onde uma anotação de tempo real pode ser definida em um modelo UML e também uma regra para definição de prioridades. O presente trabalho prefere usar anotações de tempo real em sinais UML. Esta preferência é importante para reforçar a abordagem orientada a eventos, onde um evento e o contexto corrente determinam a operação a ser executada, desde o PIM.

Um modelo PIM que segue a especificação fUML pode ser avaliado por qualquer ferramenta que implemente esta especificação. Várias alternativas existem para prover entradas para uma ferramenta que execute fUML. No presente trabalho opta-se por usar XMI, mais especificamente arquivos ".uml" manipulados pelo *plugin* do eclipse UML2 (ECLIPSE FOUNDATION, 2010d).

Em resumo, um especialista no domínio modela um PIM, a partir dos requisitos e casos de uso do sistema e dos subsistemas, para cada subsistema usando fUML, a biblioteca de tempo e o UML *profile* MARTE – HLAM. Em seguida, ele define um modelo de teste materializando terminadores definidos no subsistema a ser testado e definindo as condições de teste, tudo usando fUML. Então, ele usa uma transformação MOF QVTO para entrelaçar os dois modelos, do subsistema e de teste do subsistema, e os executa, verificando os resultados para cada caso de uso de subsistema. Quando esta verificação tiver

sido realizada com sucesso uma vez, o especialista pode agendar verificações continuas. Quando cada subsistema estiver verificado, o especialista usa novamente a transformação MOF QVTO para entrelaçar todos os subsistemas obtendo um modelo PIM para todo o sistema. Agora, os casos de uso de sistema são verificados com a mesma abordagem usada para os subsistemas. Os principais conceitos usados durante a modelagem do PIM são exibidos na Figura 3.2.

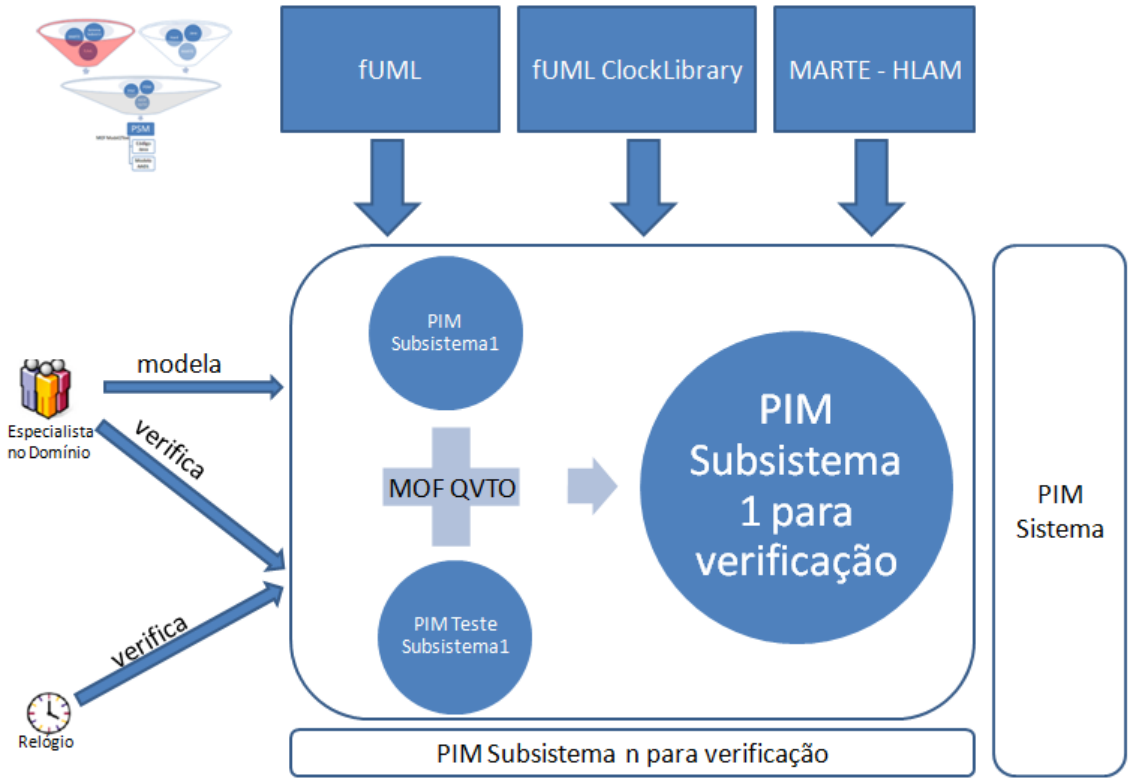

Figura 3.2 – Detalhamento da arquitetura PIM.

A próxima seção explora a arquitetura definida para o PDM.

3.3.2. PDM

O PDM prove um conjunto de conceitos técnicos, como classes ou serviços disponibilizados por uma plataforma. Como explorado brevemente na seção “3.1 Considerações para concepção da proposta”, ele deve ser subdividido em

pelo menos duas partes (esta divisão segue o Princípio da Equivalência entre Liberação e Reuso):

1. PDM Hardware – que define as características da plataforma de hardware alvo.
2. PDM Software – que define os principais conceitos e serviços da plataforma de software;

O modelo PDM Hardware usa o pacote *Hardware Resource Modeling* (HRM) do UML *profile* MARTE. Este pacote do UML *profile* MARTE provê mecanismos para modelar a parte hardware de um sistema embarcado (OMG, 2009a). É um *profile* extenso, mas a presente proposta usa apenas a capacidade de definir os principais componentes de hardware embarcados descrevendo sua estrutura. A Figura 3.3 mostra um diagrama de classes usando o *profile* HRM para modelar os componentes de hardware do Subsistema de Controle de Atitude e Órbita da PMM.

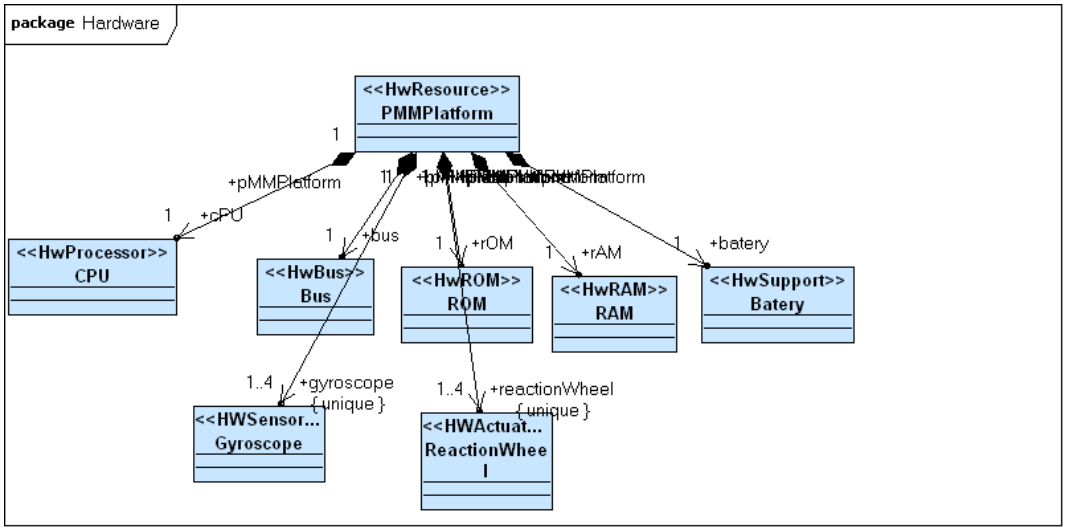

Figura 3.3 – Exemplo do uso do MARTE – HRM.

O PDM Software é composto basicamente por classes e tipos básicos da plataforma selecionada; no presente trabalho, Java, essenciais para modelagem de outros sistemas, como: *java.lang.Object, java.lang.Thread*, etc... Além disso, ele pode conter a definição de um comportamento a ser

compartilhado por toda uma plataforma. Para isso, são utilizados *OpaqueBehaviors* da UML. A UML declara *OpaqueBehavior* como sendo um comportamento com semântica específica para uma determinada plataforma (OMG, 2009b).

O PDM Software usa dois pacotes do UML *profile* MARTE, sendo eles:

- GRM – *Generic Resource Modeling* – focado em oferecer conceitos que são necessários para modelar uma plataforma geral para execução de sistemas de tempo real embarcados (OMG, 2009a);
- SRM – *Software Resource Modeling* – tipicamente usado para descrever uma API multi-tarefa de forma genérica (OMG, 2009a).

Uma definição essencial em sistemas de tempo real, que seguem a abordagem assíncrona, é sobre as características do Escalonador (*Scheduler)* provido pela plataforma de software. No MARTE, isto é definido usando o estereótipo *Scheduler* do pacote MARTE - GRM.

Além disso, é importante recuperar dois pontos não explicitamente declarados pela fUML, sendo eles:

1. A semântica de simultaneidade;
2. A semântica para o mecanismo de comunicação entre objetos.

Quanto à semântica de simultaneidade, a presente proposta assume execução simultânea para objetos de classes marcadas como ativas, ou seja, a chamada entre estes objetos é assíncrona; as outras são síncronas. Isto vale para o sistema final (código gerado em Java no presente trabalho), para verificação no nível PIM a semântica de simultaneidade é definida pela ferramenta capaz de avaliar a fUML selecionada.

Quanto ao mecanismo de comunicação entre objetos, a presente proposta define um modelo especialmente para esta finalidade (esta divisão segue o

Princípio da Equivalência entre Liberação e Reuso). O objetivo com a separação deste modelo é permitir que uma mesma plataforma de software possa chavear entre mecanismos de comunicação. Este modelo contém uma ou mais classes Java responsáveis pela comunicação entre objetos com operações UML associadas a *OpaqueBehaviors* contendo código na linguagem selecionada; na presente proposta, Java. Isto vale para o sistema final (código gerado em Java no presente trabalho), para verificação no nível PIM o mecanismo de comunicação é definido pela ferramenta capaz de avaliar a fUML selecionada.

Um modelo PDM não segue uma especificação para execução, como a fUML que permite o teste a partir de modelos. Assim, o teste do PDM só pode ser efetuado em conjunto com a transformação para PSM (detalhada na próxima seção).

Em resumo, um especialista de software modela um PDM usando os pacotes GRM, SRM e HRM do UML *profile* MARTE e *OpaqueBehaviors*. Em seguida, para arquiteturas similares, ele reutiliza o PDM previamente definido. Os principais conceitos envolvidos são exibidos na Figura 3.4.

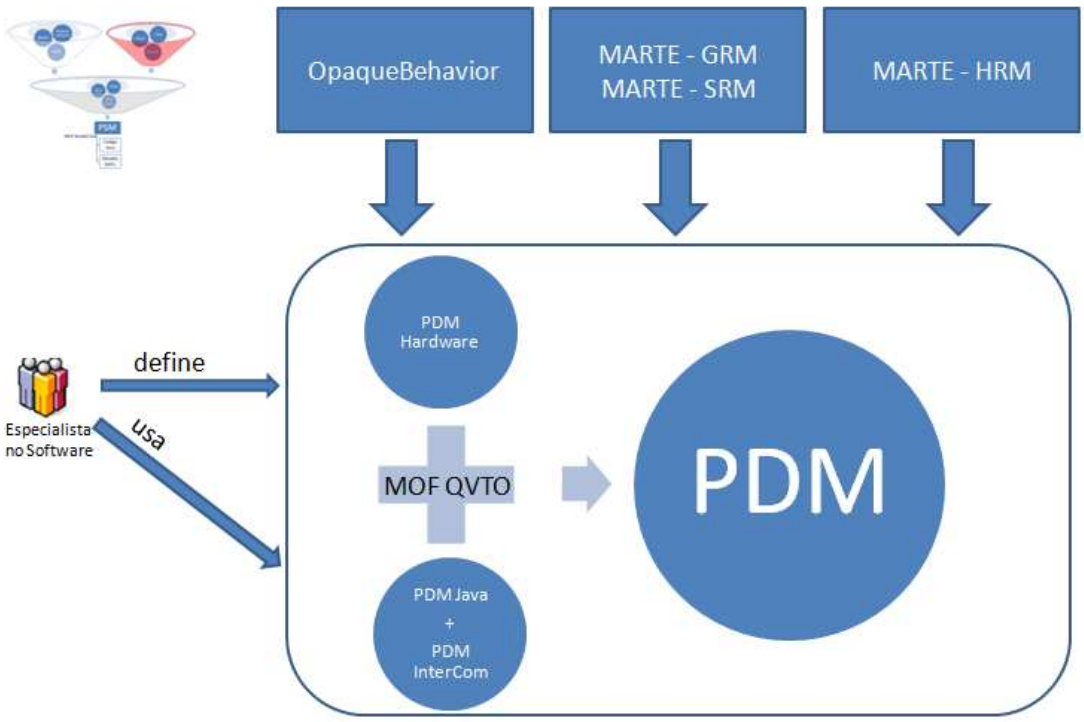

Figura 3.4 – Detalhamento da arquitetura PDM.

Os modelos PDM são entrelaçados usando-se a mesma transformação MOF QVTO citada anteriormente, responsável por entrelaçar dois modelos PIMs.

A próxima seção irá explorar a arquitetura definida para o PSM.

3.3.3. PSM

Neste ponto existe um modelo PIM, composto pelo entrelaçamento de diversos PIMs, através de uma transformação MOF QVTO. Existe também um modelo PDM, composto pelo entrelaçamento dos modelos PDM, usando-se a mesma transformação MOF QVTO.

Optou-se por entrelaçar o modelo PIM resultante com o PDM resultante, gerando um terceiro modelo que simplesmente contém todos os elementos do PIM e do PDM. Este modelo é o principal parâmetro de entrada para uma

segunda transformação MOF QVTO. Esta transformação é responsável por interpretar os dados do modelo de entrada e gerar um modelo PSM.

A transformação é um complemento dos modelos PDM, pois ela é a responsável por traduzir conceitos PIM em conceitos descritos no PDM, um exemplo didático é: no modelo PIM um campo numérico é definido utilizando-se o tipo *UML::PrimitiveType::Integer*. Em uma determinada plataforma ele deve ser transformado em um tipo básico de Java chamado *int* (definido no PDM). No presente trabalho, tal transformação é codificada usando-se a especificação MOF QVTO.

Um exemplo mais complexo é a avaliação das atividades UML (criadas usando-se diagrama de atividades no presente trabalho) e sua transformação em código na plataforma de destino selecionada, na presente proposta Java. Este tipo de transformação é difícil de ser implementada usando-se o formalismo da especificação MOF QVT. Para isso, tal especificação define o tipo de mapeamento chamado *blackbox* (OMG, 2008a). Este tipo de mapeamento é definido usando-se a linguagem Java. No presente trabalho este tipo de mapeamento foi usado principalmente na tradução das atividades fUML em código Java. Seguindo a mesma abordagem implementada no PDM, o código Java gerado a partir das atividades é armazenado no modelo PSM usando-se *OpaqueBehaviors*. Mais detalhes sobre este mapeamento são explorados no próximo capítulo.

Um ponto importante na modelagem de sistemas é permitir a rastreabilidade entre os modelos. A transformação sugerida no presente trabalho cria objetos UML chamados *Abstraction*, definidos como: uma abstração é um relacionamento que relaciona dois ou mais elementos que representam o mesmo conceito em diferentes níveis de abstração ou a partir de pontos de vista distintos (OMG, 2009b). Estes objetos documentam a rastreabilidade entre conceitos do modelo PIM para o modelo PSM.

O último item, do ponto de vista de arquitetura, a ser apresentado é: avaliar os requisitos não funcionais definidos no PIM e transformá-los em código e informações no PSM. Destaca-se aqui o uso do pacote SRM do UML *profile* MARTE para anotar automaticamente classes geradas nesta transformação. Um exemplo importante é: cada classe ativa no PIM se transforma em uma classe com o estereótipo *SwSchedulableResource* cujos atributos prazo e período são obtidos nas NFPs definidas no PIM. Outro exemplo é associar a classe principal do sistema, *MemoryPartition* no SRM, a um dispositivo físico de memória. Convêm frisar que esta abordagem para a definição de tarefas é apenas um exemplo de uma transformação focada em uma arquitetura e plataforma. Na literatura estão disponíveis critérios para estruturação de tarefas (Gomaa, 2000).

Assim a engenharia de uma plataforma é composta pela definição de um PDM e da transformação que é responsável por avaliar um PIM, considerar o PDM, e gerar como saída um PSM.

Uma vez que o PSM foi gerado, o especialista em software complementa o modelo com informações adicionais. Um exemplo importante é a definição do tempo de ativação de uma classe anotada automaticamente com o estereótipo *SwSchedulableResource*. Esta informação é crucial para a avaliação de escalonabilidade, mas não é definida como uma NFP no PIM, pois depende de vários fatores, entre eles destacam-se: código gerado, a plataforma de software e de hardware.

O PSM só pode ser verificado depois que ele for serializado transformando-se em código. Esta serialização é detalhada na próxima seção. Usualmente define-se um pequeno modelo PIM, aplica-se a transformação para PSM e gera-se o código, que então é avaliado em detalhes. Este ciclo é executado até que a transformação PSM seja estabilizada. É importante destacar que, se um

erro for localizado em um PDM ou em uma transformação (para PSM), tal erro estará replicado por todos os sistemas gerados pela plataforma (PDM + transformações) em questão. Conseqüentemente, se um erro for corrigido em uma plataforma, então todos os sistemas podem ser corrigidos com a nova geração do código.

Em resumo, um especialista de software codifica, testa e usa uma ou mais transformações do PIM para o PSM considerando o PDM. Para isso ele usa a especificação MOF QVTO, bibliotecas *blackbox*, *OpaqueBehaviors* e o pacote SRM do UML *profile* MARTE. Os principais conceitos envolvidos são exibidos na Figura 3.5.

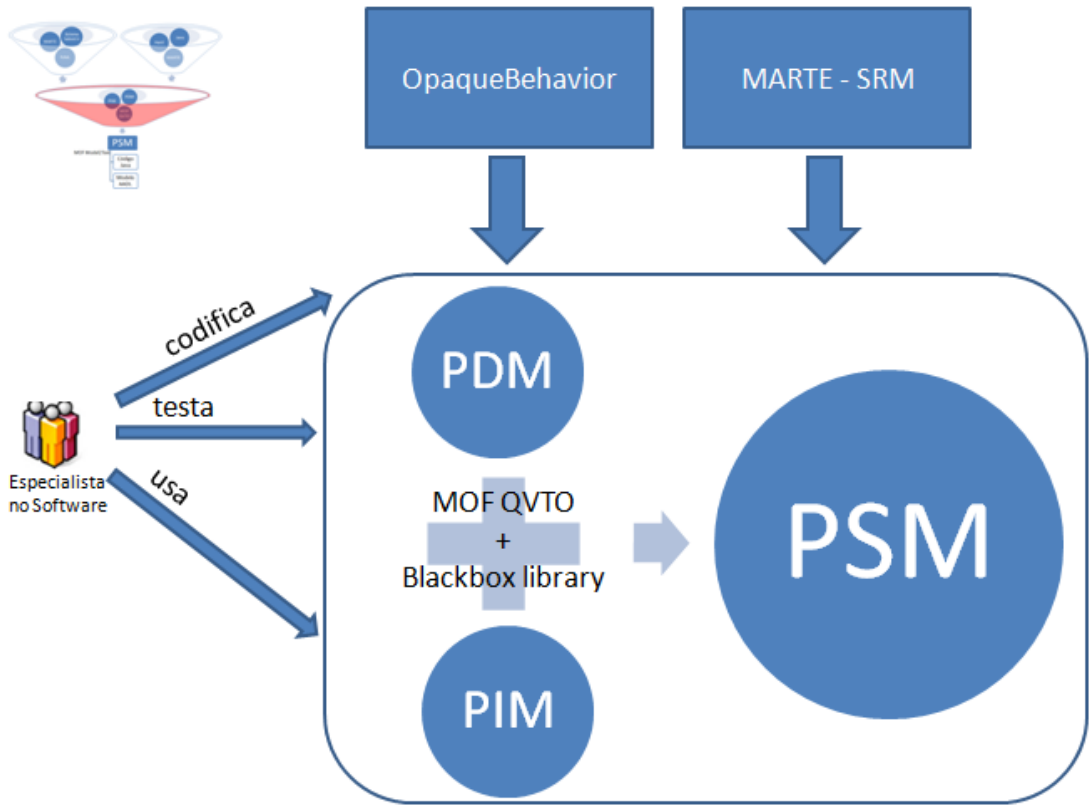

Figura 3.5 – Detalhamento da arquitetura PSM.

A próxima seção irá explorar a arquitetura definida para a geração de código.

3.3.4. Código

Para gerar o código para a plataforma de software selecionada, na presente proposta Java, uma transformação MOF M2T (OMG, 2008b) é aplicada ao PSM. Cabe aqui destacar, que esta geração de código é simplesmente uma "serialização" do modelo, já que o modelo contém todo o código de comportamento usando-se *OpaqueBehaviors* e já faz referência a classes e tipos básicos da plataforma.

O código automaticamente gerado contempla toda a regra de negócio e adequações à plataforma selecionada. Assim, ele não precisa ser modificado. Entretanto, podem existir casos onde a customização do código seja necessária. Para isso, a especificação MOF M2T (OMG, 2008b) provê meios para adição de código sem comprometer as subseqüentes execuções de transformações. Aqui é importante destacar que a ausência de interface visual em sistemas embarcados (PASETTI e BROWN, 2002) garante a geração de 100% do código necessário; e que, uma eventual alteração deveria ser realizada na transformação de PIM para PSM e não no código final.

3.3.5. AADL

No presente trabalho, um modelo AADL é crucial para a avaliação de escalonabilidade. O presente trabalho optou por transformar o PSM (um modelo UML anotado automaticamente com o UML *profile* MARTE) em um modelo AADL, mesmo que parcial, mas que permitisse a avaliação de escalonabilidade em qualquer ferramenta capaz de interpretar modelos AADL. Outros trabalhos na literatura (MAES, 2007) (MAES e VIENNE, 2007) preferem transformar um modelo UML anotado com o UML *profile* MARTE em um modelo XML específico para uma determinada ferramenta para então realizar a avaliação de escalonabilidade nesta ferramenta. Outra diferença na abordagem

empregada no presente trabalho versus Maes (2007) e Maes e Vienne (2007) é sobre quais pacotes do MARTE utilizar. Enquanto os outros trabalhos usam o pacote SAM (*Schedulability Analysis Modeling*), o presente trabalho usa os pacotes GRM, SRM e HRM. O motivo para a seleção desta abordagem diferente é descrever a estrutura do sistema, e não cenários de execução (objetivo do pacote SAM).

Para gerar um modelo AADL uma transformação MOF M2T (OMG, 2008b) é aplicada ao PSM. Cabe aqui destacar, que esta geração é simplesmente uma "serialização" do modelo, já que o modelo contém toda a informação necessária. Neste caso, informações de tarefas e as informações de período, prazo e tempo de ativação para cada uma.

3.4. Ferramentas selecionadas

As ferramentas a serem utilizadas têm uma importância crucial no presente trabalho, dado o número de especificações envolvidas, pois não existe tempo hábil em uma dissertação para implementar este volume de especificações.

Para a seleção das ferramentas foram estabelecidos os seguintes critérios:

- Mandatoriamente devem disponibilizar as seguintes especificações:
    - MOF 2.0;
    - UML 2.1 ou superior;
    - MOF-QVT Operational 1.0 ou superior;
    - MOF-M2T 1.0 ou superior;
    - MARTE profile 1.0 ou superior;
    - fUML beta 2 ou superior;
- Preferencialmente soluções abertas (*opensource*), permitindo a contribuição com a comunidade;
- Preferencialmente sem custo, devido às limitações orçamentárias de uma dissertação.

Foram avaliadas as seguintes ferramentas:

- Rational Software Architect 7.5 – IBM (IBM, 2010);
- MagicDraw 16.6 – NoMagic (NOMAGIC, 2010);
- TOPCASED 3.3 (TOPCASED, 2010);
- Eclipse Galileo Modeling (ECLIPSE FOUNDATION, 2010b);
- Cheddar 2.1 (LISYC, 2010);
- fUML Reference Implementation 0.4 (MODELDRIVEN, 2010).

Seguindo os critérios estabelecidos, as ferramentas selecionadas e assim como os *plugins* usados estão detalhados na Tabela 3.1.

Tabela 3.1 – Ferramentas selecionadas versus especificações

Ferramenta	**Usada para**	**Especificações disponibilizadas**
TOPCASED 3.3	• Modelar PIM usando UML; • Modelar PDM usando UML.	• MOF 2.0 através do *Eclipse Modeling Framework* (EMF) (ECLIPSE FOUNDATION, 2010c); • UML 2.1 através do *plugin* UML2 (ECLIPSE FOUNDATION, 2010d); • MARTE 1.0 através do *plugin* Papyrus (CEA, 2010).
Eclipse Galileo Modeling	• Implementar transformações usando MOF QVTO e MOF M2T;	• MOF QVTO 1.0 através do *plugin* MOF-QVT Operational (ECLIPSE FOUNDATION, 2010d);

	• Implementar *blackbox library*; • Analisar e executar o código gerado.	• MOF M2T através do *plugin* Acceleo (OBEO, 2010); • MARTE 1.0 através do *plugin* Papyrus (CEA, 2010).
fUML Reference Implementation 0.4	• Verificar aderência de modelos PIM a fUML; • Executar modelos PIM; • Avaliar resultados dos testes modelados.	• fUML beta 2;
Cheddar 2.1	• Avaliar escalonabilidade de modelos AADL.	• AADL;

4 IMPLEMENTAÇÃO

Este capítulo apresenta uma implementação para a arquitetura detalhada anteriormente. Primeiro os modelos PIM são abordados, apresentando-se desde a forma como eles são construídos até a avaliação dos casos de uso de subsistema. Em seguida, cada uma das outras partes, que compõem a arquitetura proposta, são detalhadas.

4.1. Funcionamento da arquitetura

Analisando-se uma situação onde não existem subsistemas PIMs previamente modelados e uma plataforma alvo definida, a Figura 4.0 descreve o funcionamento da arquitetura proposta. Nela cada atividade faz referência a uma seção do presente capítulo, permitindo ao leitor uma navegação rápida no presente capítulo.

Especial ênfase foi dada às atividades a serem realizadas pelos profissionais envolvidos, sendo elas:

- Especialista no domínio - é responsável por modelar os subsistemas e testá-los. Uma vez que todos os subsistemas estão modelados e testados (condição no diagrama de atividades), o especialista no domínio é responsável por testar o sistema (condição no diagrama de atividades); sua participação termina quando um PIM testado para o sistema todo é entregue para o especialista de software;
- Engenheiro de software - é responsável por definir a plataforma alvo, contendo PDM – Hardware, PDM – Software, transformação PIM para PSM e transformação PSM para código (a transformação PSM para AADL é mandatória somente se a plataforma alvo usa a abordagem assíncrona para descrever sistemas de tempo real); ele ainda é responsável por traduzir o PIM em PSM, avaliar a escalonabilidade até que ela seja factível (condição no diagrama de atividades), e então gerar o código final.

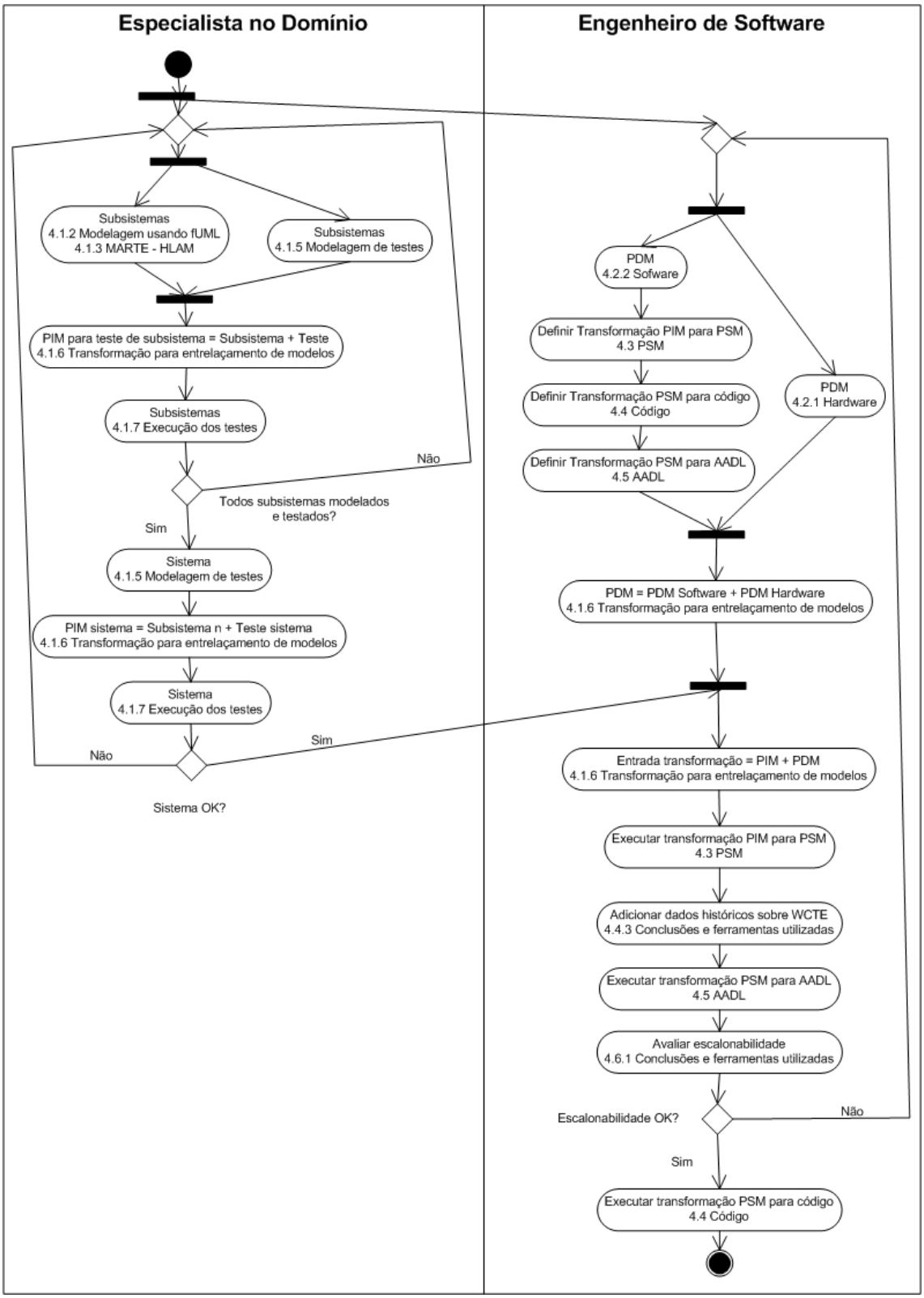

Figura 4.0 – Funcionamento da arquitetura.

4.2. PIM

Conforme já abordado, a MDA está focada no reuso de conhecimento materializado em modelos, onde cada subsistema, *domain* segundo Mellor e Balcer (2002), faz parte da propriedade intelectual de uma companhia, e as pontes, *bridges*, colocam tal propriedade intelectual para trabalhar.

Inicialmente será apresentado como UML pode descrever uma máquina de estados. Em seguida, será apresentada a forma utilizada para modelagem de cada subsistema usando fUML, considerando o uso de atividades UML e o formato XMI para avaliação por uma ferramenta capaz de interpretar fUML. Após isto será apresentada, a semântica para tempo implementada através da *ClockLibrary*. Então o uso do MARTE nos modelos PIM será explorado; e, por fim, o entrelaçamento dos modelos e os testes dos mesmos serão apresentados.

4.2.1. UML e máquinas de estado

Os diagramas de máquina de estados são uma técnica conhecida para descrever o comportamento de um sistema (FOWLER, 2005). Eles são compostos por dois conceitos básicos:

- Estado – um estado pode ser descrito por uma das alternativas abaixo (HOLT e PERRY, 2008):
    - Situações em que o sistema satisfaz uma condição particular, em termos de valores de propriedades ou eventos que ocorreram. Como, por exemplo, “inicializado” no caso de sensores do sistema de controle de atitude da PMM;
    - Uma situação onde o sistema está fazendo algo. Supõe-se que estados usam um tempo finito para execução enquanto que transições são instantâneas;
    - Uma situação onde o sistema não faz nada ou está esperando que um evento ocorra. Como, por exemplo, quando se define que

a leitura do sensor de velocidade angular da PMM será realizada de tempos em tempos, um estado de espera pela próxima leitura surge naturalmente.

- Transição – indica um movimento de um estado para outro (FOWLER, 2005).

Nos diagramas de máquina de estados um estado pode ser descrito por atividades ou ações, já transições só podem ser descritas por ações. (HOLT e PERRY, 2008) define atividade e ação como: atividade não é atômica e pode ser interrompida, dado, que elas usam um tempo finito para sua execução; ações, por outro lado, são atômicas e como tal não podem ser interrompidas, além de serem conceitualmente instantâneas.

Para a modelagem por Diagramas de Máquinas de Estados existem duas abordagens (HOLT e PERRY, 2008):

- Baseada em atividades – nesta abordagem os diagramas são criados com ênfase nas atividades realizadas nos estados. Os nomes dos estados nesta abordagem tendem a possuir verbos no gerúndio, como: Inicializando ou Lendo;
- Baseada em ações – aqui os diagramas são criados com ênfase em ações ao invés de atividades. Ações podem ser usadas tanto em transições como em estados. Com esta abordagem a tendência é usar ações nas transições, o que gera estados vazios. Os nomes dos estados nesta abordagem refletem os valores do sistema em determinado momento, como: Parado ou Ligado.

Holt e Perry (2008) ainda fornecem uma comparação entre as duas abordagens sendo os principais pontos a favor da abordagem baseada em atividades:

- A abordagem baseada em atividades é uma abordagem mais rigorosa;

- Para sistemas de tempo real, a abordagem baseada em atividades é melhor, pois não possui a premissa de que estados e transições são instantâneos.

O presente trabalho utiliza a abordagem baseada em atividades.

Dado que os Diagramas de Máquina de Estados podem ser baseados em atividades, é preciso entender qual o relacionamento deles com os Diagramas de Atividades. Na UML 1, os diagramas de atividades eram vistos como casos especiais dos diagramas de máquina de estados. Na UML 2 tal vinculo foi eliminado (FOWLER, 2005). Os diagramas de atividades são uma técnica para descrever lógica de procedimento, processo de negócio e fluxo de trabalho (FOWLER, 2005). Eles possibilitam a modelagem de fluxos de dados e de controle.

A principal diferença entre diagramas de máquinas de estado e diagramas de atividades é que o primeiro descreve o comportamento de um elemento, enquanto que o segundo geralmente dá ênfase às ações executadas pelo sistema.

Uma vez que foram apresentadas as técnicas para modelagem de máquina de estados usando UML, agora é preciso avaliar como este elemento crucial para a abordagem orientada a eventos é modelado usando-se fUML.

4.2.2. Modelagem usando fUML

Segundo Mellor e Balcer (2002) um modelo executável tem basicamente três diagramas UML:

- diagramas de classe, para modelar **dados**, descritos como classes, atributos, associações e restrições;

- diagramas de máquina de estados, para modelar **controle**, descritos como estados e transições;
- diagramas de atividades, para modelar **algoritmos**, descritos com ações.

Entretanto, o único tipo de comportamento definido por usuário suportado pela fUML é a atividade (OMG, 2009c). Por outro lado, atualmente a UML só prove o diagrama de atividades para modelagem de atividades (OMG, 2009c); e, conseqüentemente utilizando-se fUML o número de diagramas cai para dois, sendo eles:

- diagramas de classe, para modelar **dados**, descritos como classes, atributos e associações;
- diagramas de atividades, para modelar **controle** e **algoritmos**.

A especificação fUML (OMG, 2009c) declara que elementos UML como estruturas compostas e máquinas de estados não possuem uma semântica única para tradução, assim elas foram excluídas da fUML.

Para suprir a falta dos diagramas de máquinas de estado, cruciais na abordagem orientada a eventos, o presente trabalho optou por modelar máquinas de estados através dos diagramas de atividades, descrevendo estados como atividades estruturadas, *StructuredActivity*, e as transições como ações de recepção de sinal (*AcceptEventAction*) ou término da atividade contida em uma *StructuredActivity*. Esta alternativa está alinhada à abordagem baseada em atividades descrita anteriormente. A Figura 4.1 exibe um diagrama de atividades representando uma máquina de estado com a abordagem baseada em atividades.

Para cada classe ativa, ou seja, que necessita da gestão de estados, o presente trabalho prega um diagrama de atividades conforme descrito no parágrafo anterior. Este comportamento, UML *Activity*, deve ser informado no

atributo *ClassifierBehavior* de uma classe ativa. Uma validação simples é garantir que toda classe ativa possua uma atividade no atributo *ClassifierBehavior*. Classes passivas não devem possuir o atributo *ClassifierBehavior* preenchido. Neste momento, está sendo modelado o **controle** para as classes ativas. A definição de *ClassifierBehavior* seguindo a especificação UML (OMG, 2009b) é: uma especificação de comportamento que especifica o comportamento do próprio elemento.

Cada operação (UML *Operation*), tanto em classes ativas como em passivas, deve possuir o atributo *Method* preenchido com uma UML *Activity*. Não é comum operações em classes ativas, já que seu principal objetivo é fazer a gestão de estado. Algoritmos usualmente ficam em classes passivas. Neste momento estão sendo modelados os **algoritmos** para cada operação. A definição para o atributo *Method* seguindo especificação UML (OMG, 2009b) é: uma descrição comportamental que implementa um comportamento.

Sobre como modelar fluxo de dados, a especificação MARTE (OMG, 2009a) declara que existem tradicionalmente duas formas:

- Passiva – onde a chegada do dado ao armazém de dados (*data store*) não dispara o comportamento; além disso, o uso do dado não o remove do armazém de dados;
- Ativa - onde a chegada do dado ao armazém de dados dispara algum comportamento; além disso, o dado não é armazenado localmente.

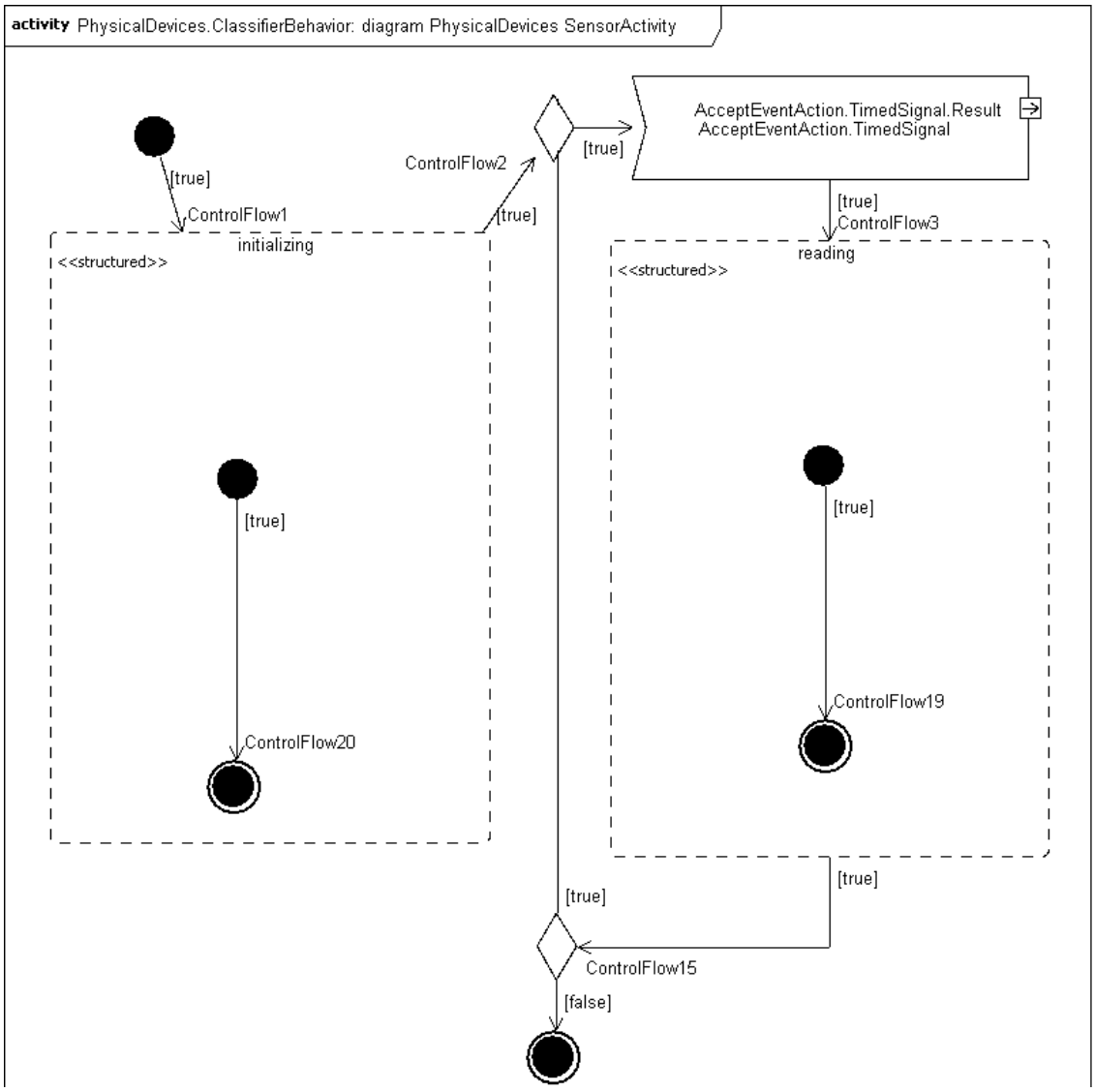

Figura 4.1 – Diagrama de atividades representando uma máquina de estados.

Uma exclusão importante na fUML, ligada à modelagem de fluxo de dados, é o elemento *CentralBufferNode*, definido na UML (OMG, 2009b) como: um nó para objetos focado na gestão de fluxos a partir de múltiplas fontes e destinos. A justificativa para sua exclusão da fUML (OMG, 2009c) é porque eles foram avaliados como desnecessários para a completude computacional da UML.

O presente trabalho usa primariamente a forma ativa, com sinais disparando comportamento; a forma passiva também é usada em alguns casos, com o armazenamento de informações em atributos de objetos.

Assim os aspectos comportamentais podem ser modelados; resta ainda modelar os aspectos estruturais. Os aspectos estruturais são modelados utilizando-se diagramas de classes, onde são modelados os dados.

Sobre classes, é importante destacar uma exclusão realizada na fUML. A UML *Interface* está fora do escopo da fUML pois, segundo a especificação (OMG, 2009c), o efeito obtido com as interfaces pode ser alcançado através de classes com todas as operações abstratas.

Os blocos elementares da fUML foram apresentados no capítulo 2. Antes de prosseguir, é preciso discorrer mais sobre o bloco *ReadExtent*. Ele é usado para localizar instâncias de um classificador e de seus filhos. É uma ação essencial para desacoplamento entre subsistemas, pois não é preciso possuir uma referência ao outro subsistema, mas sim ao terminador. Em tempo de execução são identificadas todas as instâncias de determinado terminador. Apesar de fundamental, esta ação não tem uma tradução tranqüila em sistemas de tempo real pois são naturalmente aderentes a mecanismos de persistência como um banco de dados relacional.

Agora é preciso detalhar a abordagem:

- Com relação a terminadores:
    - Todos os terminadores devem ser modelados como classes abstratas em um pacote adequado;
    - Tais terminadores devem possuir apenas um nome. É proibida a leitura de propriedades ou chamada de operações em terminadores. O objetivo é diminuir ao máximo o acoplamento.

Tal acoplamento se dará apenas através dos sinais trocados entre subsistemas;

- Só é possível a comunicação entre subsistemas através de sinais. Os sinais só podem ser enviados para classes ativas. Tais classes ativas devem declarar o sinal a ser recebido como um *Reception*;
- É proibido o uso de links entre subsistemas. Links são permitidos apenas dentro de um mesmo subsistema (não implementados no protótipo);

- Deve-se usar a ação *ReadExtent* como ferramenta de desacoplamento entre subsistemas, permitindo que subsistemas sejam localizados usando-se apenas o nome do terminador;
- Dentro de um mesmo subsistema é possível utilizar chamadas síncronas a operações de objetos passivos;
- É mandatória a modelagem de elementos de inicialização do sistema, responsáveis pela criação de objetos e inicialização do comportamento dos objetos ativos.

Para modelagem de cálculos, o único tipo disponível na fUML é o tipo primitivo *Integer*. Na biblioteca básica da fUML (OMG, 2009c) existem operações de adição, subtração, multiplicação e divisão para este tipo primitivo. A biblioteca básica da fUML (OMG, 2009c) ainda contempla operações simples sobre *String*, *Boolean*, *UnlimitedNatural* e *Lists*. Com o intuito de apresentar uma precisão mínima de cálculo, o presente trabalho optou por multiplicar todas as constantes inteiras por 1.000.000. A solução definitiva é a criação de um novo tipo primitivo *Real*, assim como tal tipo está definido no MARTE.

Para terminar esta seção de detalhamento sobre como modelar um PIM usando-se fUML segue as Figuras 4.2, 4.3 e 4.4:

- Figura 4.2 - Diagrama de classes descrevendo uma classe ativa com uma operação e uma classe passiva;

- Figura 4.3 - Diagrama de atividades de inicialização do sistema;
- Figura 4.4 - Diagrama de atividades do comportamento de uma classe ativa.

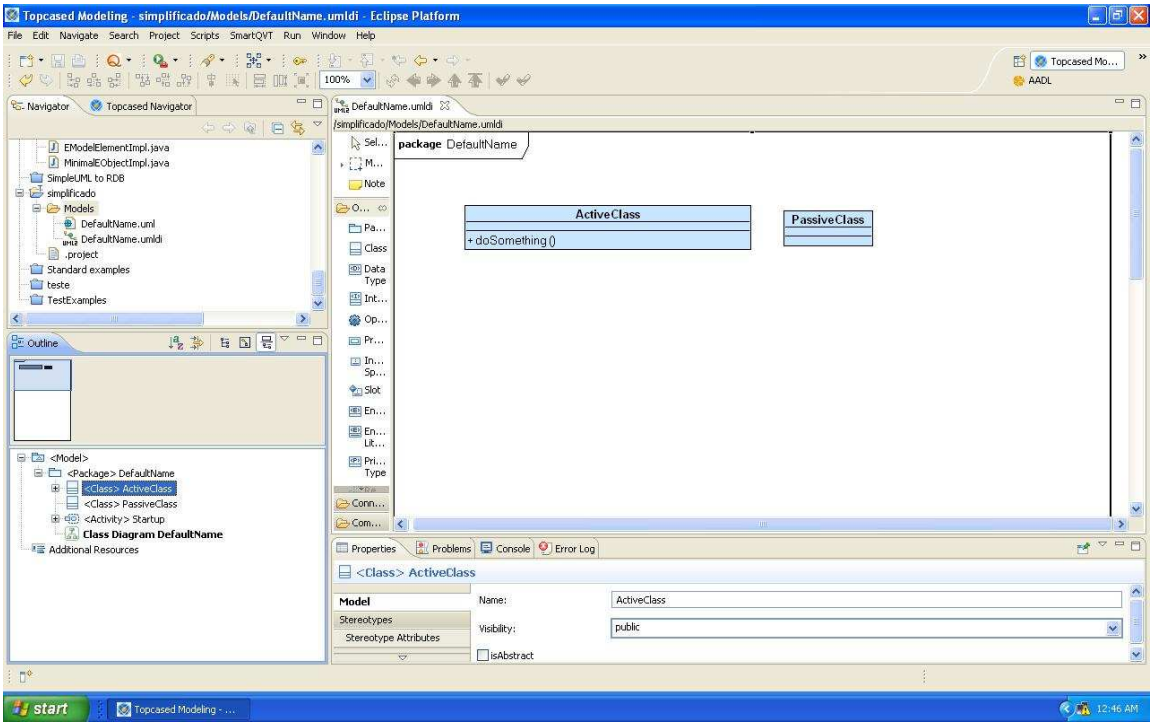

Figura 4.2 – Diagrama de classes.

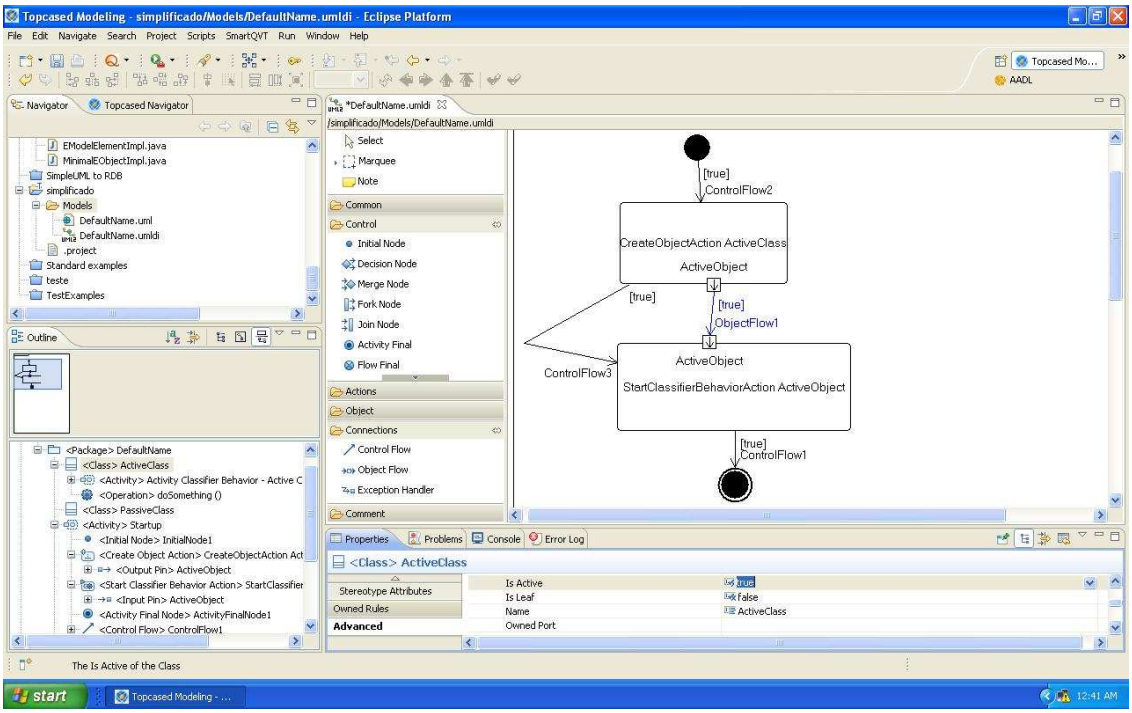

Figura 4.3 – Diagrama de atividades de inicialização do sistema.

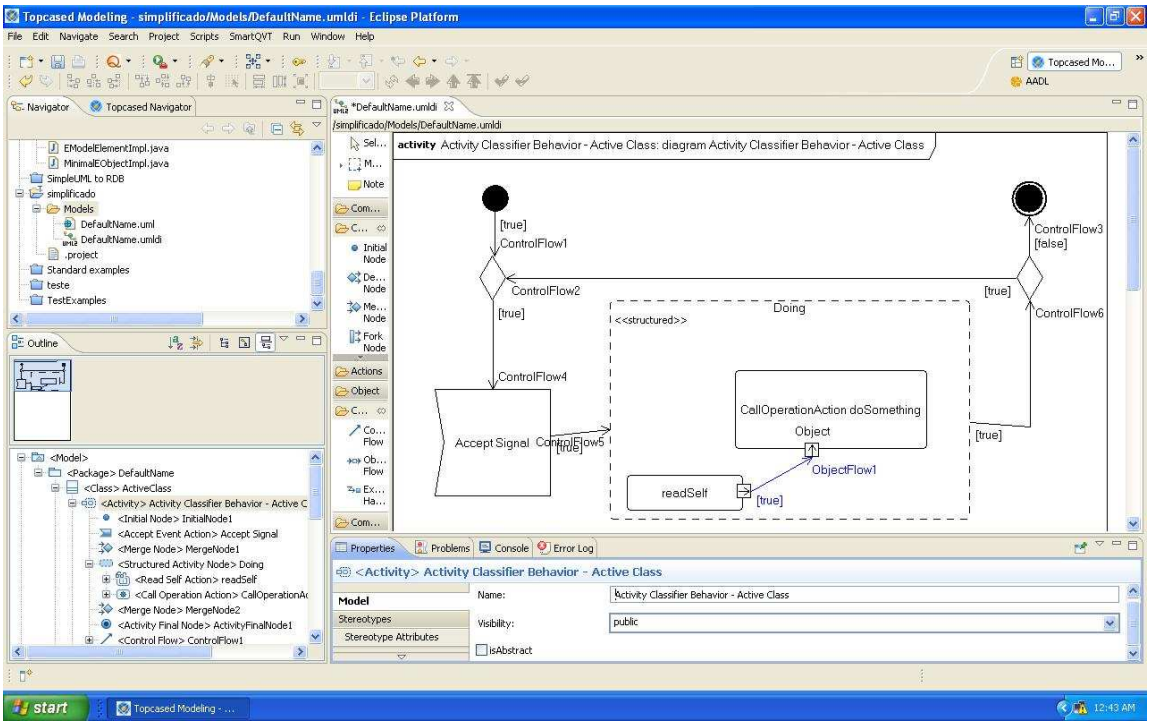

Figura 4.4 – Diagrama de atividades do comportamento de uma classe ativa.

4.2.2.1. Estruturação de subsistemas

Todos os fatores e princípios explorados no Capítulo 2 devem ser considerados na estruturação dos subsistemas, sendo que cada subsistema deve possuir um PIM independente.

Para exemplificar o uso de tais critérios para estruturação de subsistemas, a avaliação de aplicabilidade do presente trabalho, apresentada em detalhes no APÊNDICE A, avaliou o que deveria ser responsabilidade do PIM do subsistema de leitura de sensores da PMM. A principal pergunta a ser respondida sobre a estruturação deste subsistema é se a função de transferência (supondo a modelagem com Controle Clássico) para os sensores deveria ser modelada no PIM de sensores ou no PIM de controle.

Segundo Ogata (1997) a função de transferência é uma propriedade do próprio sistema, independente da magnitude ou da natureza da função de entrada. Ou seja, em qualquer local onde os mesmos tipos de sensores forem aplicados a função de transferência será a mesma, assim:

- O Princípio do Fecho Comum prega que a função de transferência deve ficar junto com a leitura dos sensores;
- O Princípio da Equivalência entre Liberação e Reuso prega que tanto a função de transferência como a leitura de sensores devem ser disponibilizados em um mesmo pacote;
- Quanto à coesão, funcionalmente, a função de transferência pertence ao controle, entretanto a coesão comunicacional é usada em toda a sua potencialidade.

A preferência pela coesão comunicacional, neste exemplo, garante alguns benefícios indiretos, como isolar a lógica para tolerância a falhas dentro do pacote de subsistema de sensores.

A seguir as propriedades não funcionais são exploradas.

4.2.3. MARTE - HLAM

Sobre como definir propriedades não funcionais em um PIM, já foi dito que o UML *profile* MARTE provê um pacote especificamente para isso, chamado *High-Level Application Modeling* (HLAM). Também já foi dito que o HLAM define vários locais onde uma anotação de tempo real pode ser definida em um modelo UML; que o presente trabalho prefere usar anotações de tempo real em sinais UML; e que a justificativa é reforçar a abordagem orientada a evento, onde um evento e o contexto corrente determinam a operação a ser executada, desde o PIM.

A Figura 4.5 exibe o uso do estereótipo *rtFeature* em um sinal contendo um comentário anotado com o estereótipo *rtSpecification*. Este último permite a definição de três atributos essenciais:

- *occKind* – especifica o padrão de geração do sinal; neste caso define que o sinal é periódico, com período de 100 milisegundos e com incerteza de liberação (*release jitter)* de 0 microsegundos;
- *miss* – determina o percentual aceitável de perda do prazo estabelecido;
- *relDl* – define o prazo relativo, sendo a referência o instante de geração do sinal.

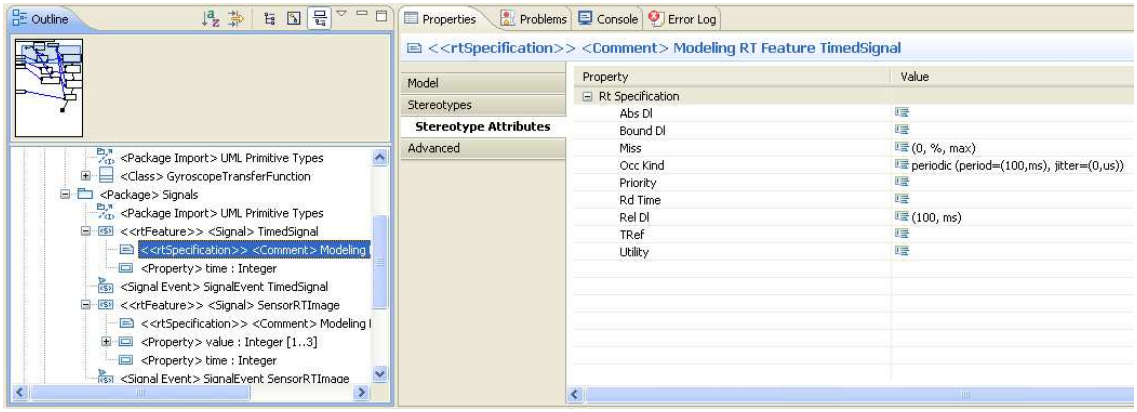

Figura 4.5 – Definição de propriedades não funcionais no PIM.

Estas propriedades não funcionais não são verificadas no modelo PIM, pois se supõe a premissa de que o tempo para cálculo e comunicação é zero (mais detalhes na próxima seção). O objetivo aqui é possuir todos os requisitos não funcionais e funcionais documentados no modelo PIM, mesmo que alguns deles não possam ser verificados no nível PIM.

Outro estereótipo usado do UML *profile* MARTE é o *RtUnit.* Segundo (OMG, 2009a) sua definição é: um *RtUnit* é similar a um objeto ativo da UML, mas com uma descrição semântica mais detalhada. No presente trabalho, e de acordo com a especificação, ele deve ser usado para anotar classes ativas no modelo PIM.

Para terminar esta seção cabe indicar que outro estereótipo do UML *profile* MARTE deve ser usado. Trata-se do *DeviceResource* do pacote GRM. Ele é usado para anotar elementos onde o comportamento interno não é relevante para o modelo em desenvolvimento (OMG, 2009a). No presente trabalho propõe-se anotar as classes responsáveis pela leitura dos sensores e ativação dos atuadores. Tais elementos envolvem chamadas a instruções ligadas à plataforma, como leitura e escrita em uma posição determinada de memória.

4.2.4. ClockLibrary

Conforme já citado anteriormente, a fUML não restringe uma única semântica para o tempo. Assim, ela permite variadas formas de tempo, incluindo tempo discreto (como modelos síncronos de tempo) e tempo contínuo (denso) (OMG, 2009c). Além disso, ela não determina premissas sobre fontes para informações de tempo e mecanismos relacionados, permitindo tanto modelos de tempo centralizados como distribuídos (OMG, 2009c).

Para reforçar o parágrafo anterior, o pacote *SimpleTime* definido na UML2 (OMG, 2009b) foi inteiramente excluído da fUML, pois eventos e restrições de tempo não fazem parte do escopo da fUML (OMG, 2009c).

A semântica de tempo para o sistema final, código gerado em Java no presente trabalho, é definida parte pela plataforma alvo e parte pela transformação dos modelos PIM para PSM. Entretanto para modelar um sistema de tempo real e verificá-lo através do modelo PIM, é preciso definir uma semântica para o tempo. Para isso, o presente trabalho propõe uma biblioteca modelada com fUML para a definição de comportamentos básicos para um relógio ainda no nível PIM. O objetivo é definir uma fonte e um mecanismo para geração de sinais baseados no tempo. Esta biblioteca é completamente ignorada pela transformação de PIM para PSM. Pois, como já foi apresentado, a semântica

de tempo para o sistema final é determinada em parte pela plataforma alvo e parte pela transformação dos modelos PIM para PSM.

Para fins de modelos PIM o presente trabalho prega que um relógio centralizado e ideal deve ser usado, para garantir independência de plataforma e permitir a simulação de eventos de tempo e também um relógio físico.

Avaliando-se o MARTE encontra-se uma classe chamada *IdealClock* e também sua instância chamada *idealClk*. Estes elementos modelam um tempo ideal que é usado em leis da Física (OMG, 2009a). Eles definem um tipo denso de tempo, definido no MARTE como: no tempo denso, para um dado par de instantes sempre existe pelo menos um instante entre estes dois (OMG, 2009a). Entretanto tal classe não pode ser utilizada, pois ela não define o comportamento para os métodos *currentTime* e *setTime*.

A Figura 4.6 ilustra a classe definida na biblioteca *ClockLibrary*.

É preciso destacar que a classe está anotada com o estereótipo *ClockType* do MARTE, e que as operações *currentTime* e *setTime* estão definidas. Além disso, a natureza está definida como densa e o relógio está marcado como não lógico, ou seja, lê um tempo cronométrico, isto é, um tempo determinado pelo tempo físico. Além das duas operações já citadas, uma terceira operação foi definida, chamada *waitTime*. Ela é responsável por incrementar o tempo do relógio seguindo um parâmetro recebido. A Figura 4.7 mostra a definição do algoritmo, usando fUML através de diagramas de atividades, para a operação *currentTime*.

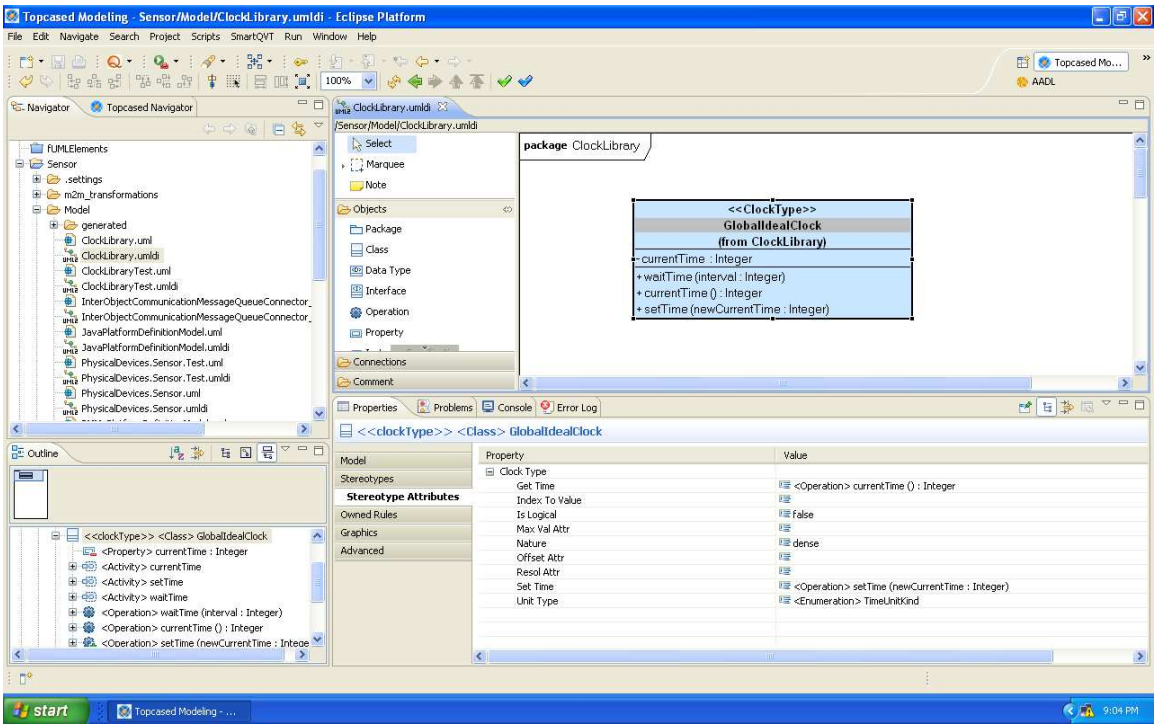

Figura 4.6 – Modelagem de estrutura da *ClockLibrary* (Diagrama de classes).

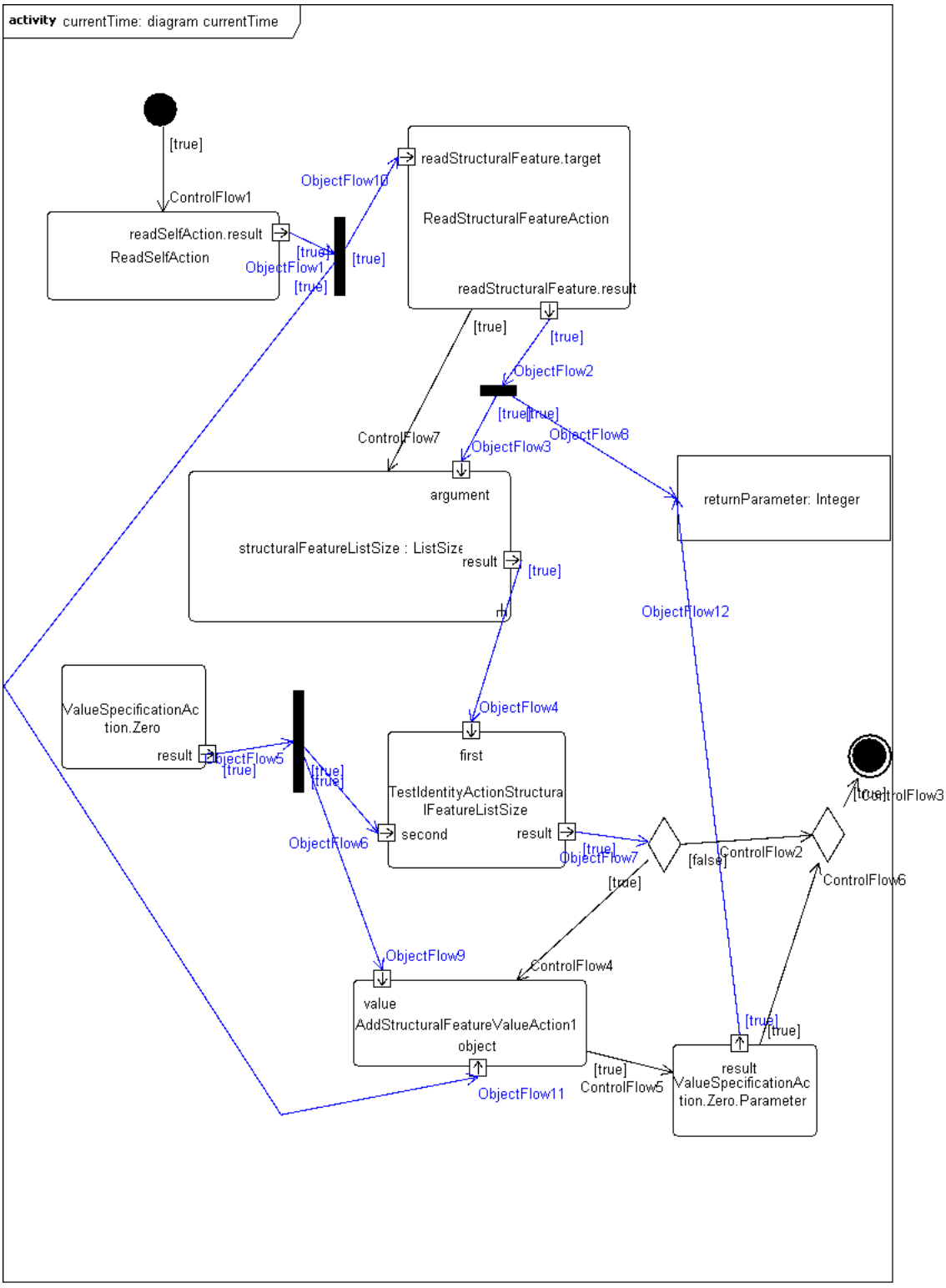

Figura 4.7 – Modelagem da operação *currentTime* do *GlobalIdealClock.*

Com estes pontos definidos, um modelo de teste para um determinado subsistema instancia um relógio, entra em um laço onde a operação *waitTime*, com a granularidade de tempo necessária para a simulação, é chamada; e então, os sinais são enviados para os terminadores usando-se como argumento o tempo corrente obtido do relógio. Este laço é executado até que um limite superior de tempo seja alcançado. A seguir, é apresentada a Figura 4.8 exibindo um diagrama de atividades que ilustra tal abordagem.

É importante destacar que, seguindo esta abordagem, a fUML assume a hipótese de que os cálculos e as comunicações não levam tempo; ou seja, a fUML é uma linguagem síncrona para modelagem de sistemas de tempo real. Ou ainda, ela é orientada ao comportamento de uma aplicação e a sua verificação. Uma vez que os modelos PIM são verificados, eles são transformados para modelos adequados à plataforma alvo, que podem seguir tanto a abordagem assíncrona como a síncrona. No presente trabalho, o modelo PIM é transformado para a plataforma Java, essencialmente assíncrona.

Sobre a visão do tempo na execução do modelo PIM por uma ferramenta capaz de interpretar fUML, segundo Mellor e Balcer (2002) deve seguir a visão relativista de Einstein do tempo. Exemplificando, se um sinal for transmitido a duas estrelas, e a primeira transmite um sinal para a segunda estrela, não é possível determinar qual sinal irá chegar primeiro na segunda estrela. Pode-se afirmar apenas que dois sinais enviados para uma única estrela irão chegar seguindo a ordem de envio.

Usando-se o MARTE – HLAM e a fUML é possível modelar os subsistemas de um dado sistema. A *ClockLibrary* é essencial para que estes subsistemas acessem o tempo corrente no modelo (mandatório para sistemas dinâmicos) e também para a modelagem dos cenários de teste. A próxima seção foca em como definir modelos para cenários de teste.

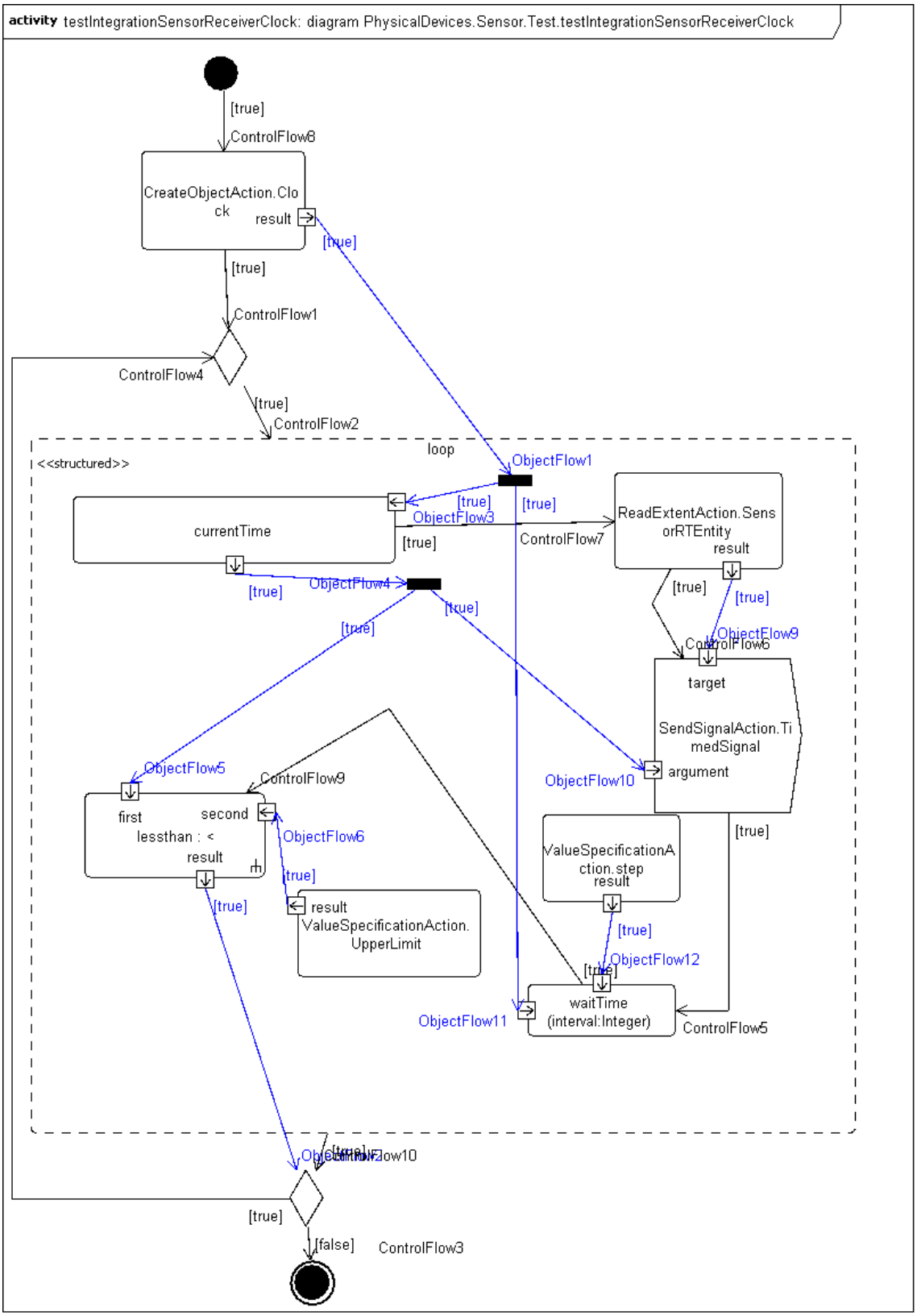

Figura 4.8 – Uso da *ClockLibrary*.

4.2.5. Modelagem de testes

Na seção anterior ao se detalhar a abordagem para uso da *ClockLibrary*, ilustrada na Figura 4.8, uma parte da abordagem para testes foi mostrada. O objetivo da corrente seção é detalhar como os modelos PIM contendo cenários de teste devem ser modelados, de acordo com o presente trabalho.

Até aqui, ao avaliar um subsistema existe um modelo PIM contendo os elementos do subsistema e seus terminadores (marcados como elementos abstratos). O modelo de teste é similar a outros subsistemas, ele possui elementos concretos e terminadores. Para que o modelo a ser testado e o modelo de teste funcionem corretamente juntos, é preciso que um terminador do modelo a ser testado seja um elemento concreto no modelo de teste, e vice-versa. Esta característica é mandatória para quaisquer subsistemas que precisem funcionar em conjunto. A conseqüência disso é que ao se entrelaçar os modelos, eles só possuíram elementos concretos. O objetivo do uso de terminadores, como abordado anteriormente, é permitir o desacoplamento entre subsistemas.

Assim um modelo de teste deve conter:

1. Elementos concretos, simuladores, para cada terminador referenciado pelo modelo a ser testado;
2. Terminadores para cada elemento concreto do modelo a ser testado;
3. Um relógio para disparar sinais para os elementos do subsistema;
4. Condições de teste.

Quanto aos itens 1 e 2, o presente trabalho assume que um terminador e um elemento concreto que materialize tal terminador possuem o mesmo nome e estão em um pacote de nome igual. O motivo pelo qual esta abordagem foi selecionada é explorado na próxima seção, dedicada especificamente a entrelaçamento de modelos. Enquanto um terminador não possui nenhum

atributo ou comportamento, um elemento concreto possui ambos. Para esclarecer este ponto a Figura 4.9 mostra um terminador em um modelo de um subsistema (esquerda) e o elemento concreto equivalente em um modelo de teste (direita).

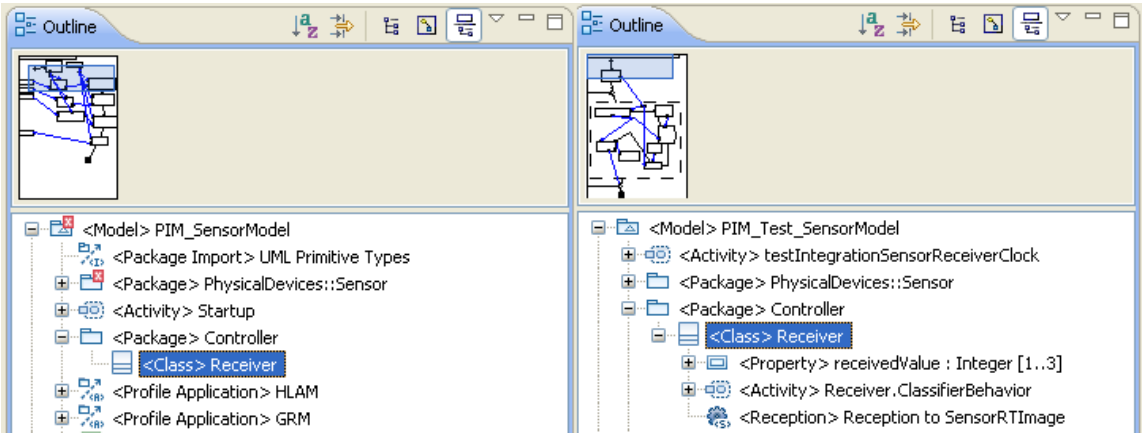

Figura 4.9 – Terminador e elemento concreto.

A partir do momento que os terminadores necessários para os cenários de teste foram modelados, que o relógio foi instanciado e configurado para disparar sinais de tempo (conforme detalhado na seção anterior) então os cenários de teste podem ser modelados. Como descrito anteriormente a fUML só permite a definição de comportamento usando-se atividades, assim os cenários de teste devem ser definidos usando-se atividades também. O presente trabalho prega que cada cenário deve ser definido em uma atividade distinta. Todos os cenários devem estar na raiz do modelo e seu nome deve começar com “*test*”. A Figura 4.10 ilustra um cenário de teste.

O motivo para que os testes estejam na raiz e comecem com “*test*” é que eles não estão ligados diretamente a pacotes e a transformação de PIM para PSM deve possuir alguma forma para identificar que esta atividade não deve ser transformada para o código final. Isto possibilita que modelos de subsistemas e seus respectivos modelos de teste possam ser transformados em código final e avaliados individualmente.

Outro ponto importante é que um cenário de teste deve sempre retornar um valor booleano calculado a partir da comparação (*TestIdentityAction*) de uma situação desejada com uma situação obtida na execução do teste. Isto é importante para que os testes sejam executados automaticamente e seus resultados indiquem o sucesso ou não da execução.

A próxima seção detalha como o modelo de um subsistema e do modelo de teste de tal subsistema são entrelaçados para posterior execução.

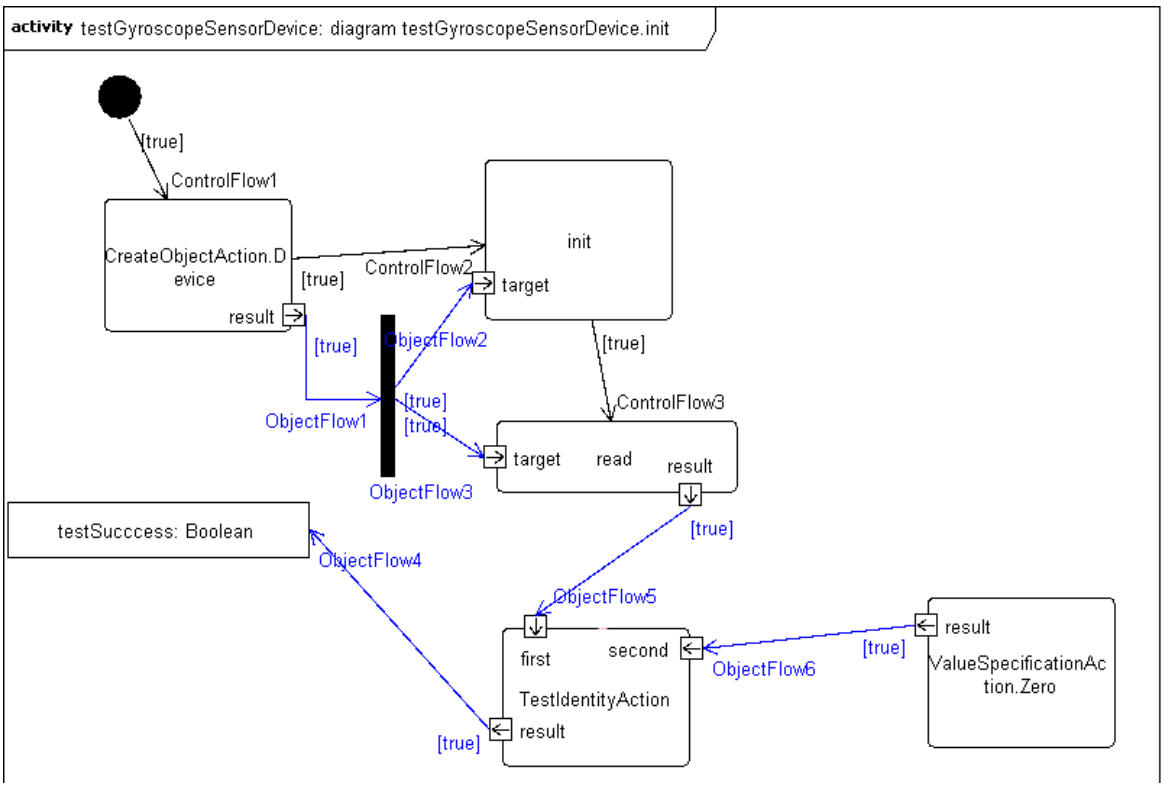

Figura 4.10 – Exemplo de um cenário de teste.

4.2.6. Transformação para entrelaçamento de modelos

Conforme apresentado anteriormente, poucos trabalhos na literatura abordam a necessidade do entrelaçamento de dois modelos (MARCOS et al., 2005),

entretanto tanto o trabalho referenciado quanto o presente trabalho consideram que tal necessidade será cada vez mais premente à medida que a MDA ganhe maturidade. Marcos et al. (2005) prega que esta necessidade nasce naturalmente quando a MDA propaga a fusão dos modelos PIM e PDM para geração do PSM, entretanto o presente trabalho advoga que para estes fins uma transformação, no sentido estrito, é mais adequada, pois não basta entrelaçar, por exemplo, substituindo a classe *UML::PrimitiveType::Integer* pelo tipo básico *int* da plataforma Java, é preciso muito mais, como a geração de código de comportamento baseado nas atividades fUML. No presente trabalho o entrelaçamento de modelos é uma ferramenta fundamental para desacoplamento, pois ela é a técnica selecionada para materializar as pontes.

Um das dificuldades descritas em Marcos et al. (2005) para o entrelaçamento de modelos é que os relacionamentos entre elementos de dois modelos não podem ser gerados automaticamente, pois são freqüentemente baseados em decisões de projeto ou heurísticas. Assim o presente trabalho optou por assumir o nome qualificado UML, *qualifiedName*, como a forma para descrever o relacionamento entre elementos de dois modelos. É uma simplificação importante para garantir que o processo de entrelaçamento seja realizado, permitindo o desacoplamento entre modelos de subsistemas.

O entrelaçamento de modelos no presente trabalho, assim como em Marcos et al. (2005), é um processo que recebe dois modelos e um meta-modelo de entrelaçamento como entrada e gera como resultado um modelo entrelaçado. No presente trabalho o meta-modelo de entrelaçamento é suprimido pela premissa de que os relacionamentos entre os elementos são descritos por seu nome qualificado.

Para implementar este processo o presente trabalho optou pela especificação MOF QVTO (OMG, 2008a), assim tal processo trata-se de uma transformação que recebe dois modelos como entrada e gera um modelo como saída. Uma

transformação MOF QVTO representa a definição de uma transformação unidirecional expressa de forma imperativa, ela define uma assinatura indicando os modelos envolvidos e define um ponto de entrada para sua execução (OMG, 2008a). A Figura 4.11 exibe a assinatura da transformação implementada, chamada de *ModelWeaver*.

```
transformation ModelWeaver( in modelIn1:UML, in modelIn2:UML, out
wovenModel:UML);

-- used to infer that abstract classes are terminators
configuration property inferThatAbstractClassesAreTerminators :
Boolean;

-- used to enable cascade weaver
configuration property allowUndefinedClassifierAndOperation : Boolean;

-- main
main() {
```

Figura 4.11 – Assinatura da transformação para entrelaçamento de modelos.

É preciso esclarecer o funcionamento da transformação, para isso o presente trabalho descreve as regras definidas para cada elemento do meta-modelo UML. É importante destacar que os dois modelos de entrada seguem o meta-modelo UML e o meta-modelo de entrelaçamento foi substituído por uma convenção, *qualifiedName* iguais. A Tabela 4.1 lista as regras usadas.

Tabela 4.1 – Regras utilizadas no *ModelWeaver*

Origem (meta-modelo UML)	**Destino (meta-modelo UML)**	**Restrição**
Profile Application	*Profile Application*	
Stereotype Application	*Stereotype Application*	
Package	*Package*	Se possuir elementos filhos
PackageableElement	*PackageableElement*	Se não for abstrato
SignalEvent	*SignalEvent*	Se sinal referenciado não for abstrato

Com tais regras cada elemento dos modelos de entrada é copiado para o modelo de saída, inclusive estereótipos aplicados aos elementos originais. A única exceção são os terminadores, que por serem classes abstratas, não são copiadas.

Após a aplicação de tais regras, existe um trabalho significativo que é corrigir as referências, removendo referências do modelo original e apontando para o modelo de saída. Por exemplo, uma atividade dispara um sinal para um terminador t1 no modelo do subsistema, tal terminador é simulado pelo elemento t1 no modelo de teste do subsistema, a transformação de entrelaçamento precisa alterar a atividade que dispara o sinal para referenciar o t1 no modelo entrelaçado.

Sobre as duas propriedades de configuração, a funcionalidade de cada uma é definida abaixo:

- *inferThatAbstractClassesAreTerminators* – usada para permitir que elementos abstratos possam ser copiados para o modelo final, importante para entrelaçamento de modelos que usam pontes implícitas, ou seja, referenciam diretamente outros modelos. Para pontes explícitas deve ser sempre verdadeiro, ou seja, assume que para cada terminador (classe abstrata) existe uma classe concreta, conseqüentemente não é possível usar classes abstratas neste caso;
- *allowUndefinedClassifierAndOperation* – usado para indicar que verificações devem ser realizadas garantindo que todos os elementos abstratos dos dois modelos de entrada foram materializados no modelo de saída; Para entrelaçamento de modelos em cascata, onde existe mais de um modelo que define terminadores, é preciso utilizar este indicador como falso.

Após o entrelaçamento do modelo de um subsistema com seu modelo de teste é possível verificar o modelo resultante usando-se uma ferramenta capaz de

interpretar fUML, tal verificação é detalhada na seção a seguir. É importante destacar que o entrelaçamento de modelo pode ser aplicado para subsistemas até se obter o sistema completo.

4.2.7. Execução dos testes

Uma vez que existe um modelo contendo tanto o(s) subsistema(s) como seu respectivo(s) modelo(s) de teste é preciso verificá-los usando uma ferramenta que seja capaz de interpretar fUML.

Conforme já apresentado no capítulo proposta, a ferramenta selecionada para verificar os modelos PIM é a fUML Reference Implementation 0.4 (MODELDRIVEN, 2010). Ela exige que o modelo de entrada esteja no formato XMI 2.1 (OMG, 2007) e que as atividades a serem executadas estejam na raiz do modelo.

Os modelos gerados pela transformação para entrelaçamento de modelos são gerados em arquivos ".uml" (formato do *plugin* UML do Eclipse), entretanto este tipo de arquivo segue o formato definido no XMI 2.1. Assim para executar o modelo a seguinte linha de comando é usada:

org.modeldriven.fuml.FUML PIMWithClock.uml Startup StartupTest testGyroscopeSensorDevice

Onde:

- *PIMWithClock* – é o resultado do entrelaçamento do modelo de um subsistema (leitura de sensores da PMM) com seu respectivo modelo de teste e ainda também com a biblioteca *ClockLibrary*;
- *Startup e StartupTest* – são as atividades de inicialização do sistema;
- *testGyroscopeSensorDevice* – é o cenário de teste a ser avaliado.

Esta execução gera o resultado exibido na Figura 4.12. Destaca-se na Figura 4.12 que a última linha indica o resultado do cenário de teste.

A partir deste momento, poderia ser agendado um processo que executa o entrelaçamento dos modelos sobre gestão de configuração, disparasse o comando exibido anteriormente executando o cenário de teste e mediante o resultado, verdadeiro ou falso, tomasse alguma ação, como disparar email para os responsáveis pela última alteração nos modelos.

```
13 Aug 2010 00:04:11,309 INFO  [main] fuml.FUML loading file, PIMWithClock.uml
13 Aug 2010 00:04:15,747 INFO  [main] config.FumlConfiguration initializing...
13 Aug 2010 00:04:15,747 INFO  [main] config.FumlConfigDataBinding loading schema chain...
13 Aug 2010 00:04:15,903 INFO  [main] library.Library initializing...
13 Aug 2010 00:04:15,903 INFO  [main] library.Library loading file, fUML_Library.uml
...
13 Aug 2010 00:04:17,106 DEBUG [main] fUML.Debug [addToken] node = second
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [removeToken] node = result
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [_endIsolation] End isolation.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [fire] Action TestIdentityAction...
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [takeTokens] node = TestIdentityAction, pin = first
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [removeToken] node = first
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [takeTokens] node = TestIdentityAction, pin = second
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [removeToken] node = second
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [putToken] node = TestIdentityAction
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [addToken] node = result
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [receiveOffer] node = testSucccess
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [_beginIsolation] Begin isolation.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [receiveOffer] Firing.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [_endIsolation] End isolation.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [fire] Output activity parameter node testSucccess...
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [addToken] node = testSucccess
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [removeToken] node = result
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [fire] Checking if TestIdentityAction should fire again...
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [_beginIsolation] Begin isolation.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [_endIsolation] End isolation.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [fire] Checking if ValueSpecificationAction.Zero should fire again...
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [_beginIsolation] Begin isolation.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [_endIsolation] End isolation.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [execute] Activity testGyroscopeSensorDevice completed.
13 Aug 2010 00:04:17,122 DEBUG [main] fUML.Debug [destroy] object = org.modeldriven.fuml.environment.ActivityExecution@17d6ac2
[execute] output parameter 'testSucccess' has 1 value(s)
[execute] value [0] = true
```

Figura 4.12 – Verificação do teste *testGyroscopeSensorDevice.*

Devido ao estágio da ferramenta fUML RI, algumas limitações foram localizadas:

- Não trabalha bem com *InputPins* e *OutputPins* de entrada e saída em atividades estruturadas (*StructuredActivity*);
- Não executa corretamente atividades que envolvam laços;

Até este ponto é possível modelar PIMs para subsistemas usando-se fUML, modelar testes para os subsistemas usando-se também fUML, executar o entrelaçamento de modelos usando-se a transformação definida, baseada na

especificação MOF QVTO, e então verificar se os cenários de teste são atendidos ou não.

4.2.8. Conclusão e ferramentas utilizadas

Antes de passar para o detalhamento do PDM é preciso indicar quais ferramentas foram utilizadas até aqui. Todos os modelos foram construídos utilizando-se a ferramenta TOPCASED 3.3 (TOPCASED, 2010). A transformação para entrelaçamento de modelos foi construída e executada usando-se a ferramenta Eclipse Galileo Modeling (ECLIPSE FOUNDATION, 2010b). A verificação dos modelos PIM foi realizada com a ferramenta fUML Reference Implementation 0.4 (MODELDRIVEN, 2010).

4.3. PDM

Conforme já apresentado a MDA prega um modelo chamado PDM (*Platform Description Model*) ou simplesmente PM (*Platform Model*). O PDM prove um conjunto de conceitos técnicos, como classes ou serviços disponibilizados por uma plataforma.

Também já foi apresentada a necessidade da divisão deste modelo em pelo menos duas partes:

1. PDM Hardware – que define as características da plataforma de hardware alvo.
2. PDM Software – que define os principais conceitos e serviços da plataforma de software;

O modelo PDM Hardware e o modelo PDM Software são entrelaçados, assim que estabilizados, usando-se a mesma transformação para entrelaçamento de modelos definida na seção anterior, gerando um modelo PDM único.

Inicialmente será apresentado como este trabalho prega a modelagem da parte do PDM focada em Hardware. Em seguida, a parte focada em software é explorada em detalhes.

4.3.1. Hardware

O modelo PDM Hardware usa o pacote *Hardware Resource Modeling* (HRM) do UML *profile* MARTE. A Figura 3.3 mostra um diagrama de classes usando o *profile* HRM para modelar os componentes de hardware do subsistema de controle de atitude da PMM. A Figura 4.13 mostra o modelo em si do PDM Hardware para o subsistema de controle de atitude da PMM.

Cada estereótipo possui uma grande quantidade de atributos focados na descrição das propriedades de cada elemento de hardware, o presente trabalho usa apenas uma propriedade do estereótipo *hwProcessor*, mais especificamente a propriedade *ownedHw*. Seguindo o MARTE (OMG, 2009a), a propriedade *ownedHW* especifica os elementos de hardware mantidos pelo *hwProcessor*. Ela é mandatória, pois será usada no momento da geração do modelo AADL, descrito na seção correspondente. Os outros atributos são opcionais.

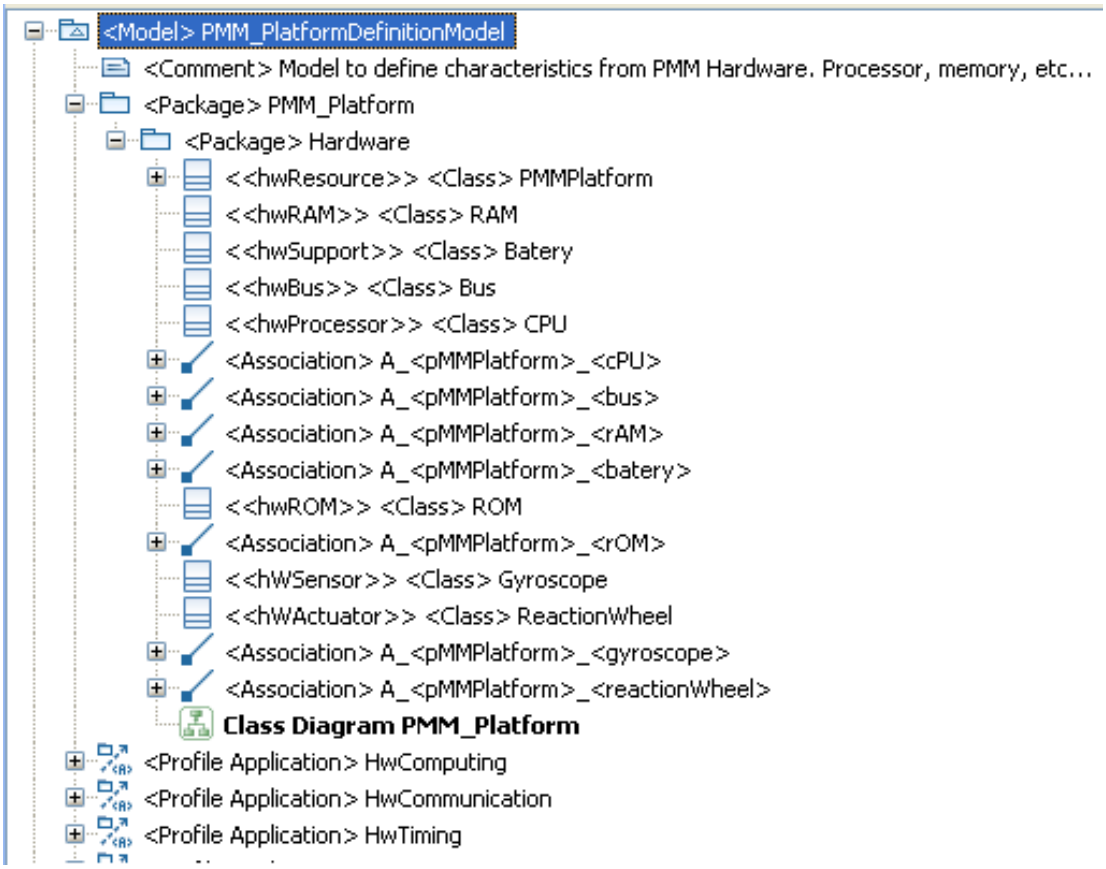

Figura 4.13 – Modelo PDM – Hardware.

4.3.2. Sofware

O PDM Software é composto por classes e tipos básicos da plataforma selecionada, no presente trabalho Java, essenciais para modelagem de outros sistemas, como: *java.lang.Object, java.lang.Thread*, etc...

A Figura 4.14 detalha claramente o conteúdo do PDM Software para uma plataforma, Java no presente trabalho. Destaca-se que não é preciso definir todos os elementos de uma plataforma, mas sim aqueles a serem utilizados na transformação PIM para PSM.

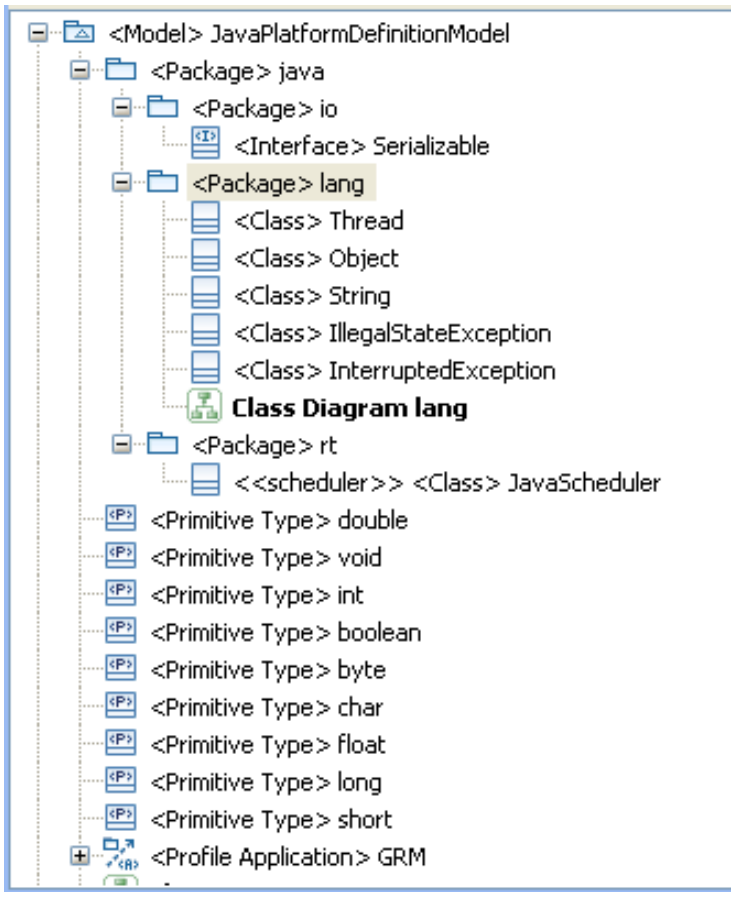

Figura 4.14 – Modelo PDM – Software.

O PDM Software usa o pacote GRM do UML *profile* MARTE. Uma definição essencial em sistemas de tempo real, que seguem a abordagem assíncrona, é o *Scheduler* provido pela plataforma de software. O MARTE define suas propriedades usando o estereótipo *Scheduler* do pacote MARTE - GRM. Como já abordado anteriormente, a linguagem Java, selecionada para fins de protótipo, não é compatível com sistemas de tempo real, e assim não oferece um *Scheduler*. Entretanto para efetuar a análise de escalonabilidade do sistema, um dos objetivos do presente trabalho, é mandatório a existência de um *Scheduler*. Assim na PDM da avaliação de aplicabilidade foi modelado um *Scheduler* preemptivo seguindo a política *Rate Monotonic* (FARINES et al., 2000) (BRIAND e ROY., 1999). A Figura 4.15 mostra estes atributos no elemento do modelo da PDM - Software.

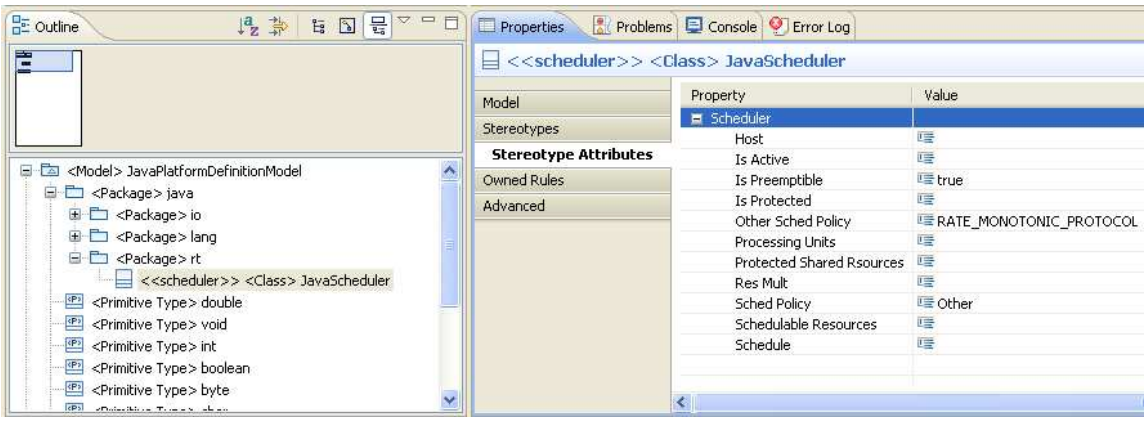

Figura 4.15 – Características do *Scheduler* na PDM - Software.

Um modelo PDM – Software pode conter outros elementos, como por exemplo, um quadro de trabalho para leitura de sensores ou para manipulação de atuadores. Neste caso, existem duas abordagens possíveis:

1. A criação de um componente de software com elementos adicionais a plataforma de software, e subseqüente modelagem dos elementos essenciais;
2. A definição deste quadro de trabalho através de modelos UML, e geração do código correspondente no momento da transformação de modelos UML para código.

Na primeira abordagem, um componente de software é criado e testado de forma isolada, então os elementos a serem utilizados pela transformação PIM para PSM são modelados na PDM – Software. Neste caso os elementos modelados não precisam conter comportamento ou atributos. No final do processo, o código gerado pela transformação PSM para código é ligada ao código do componente de software de alguma forma.

Na segunda abordagem, uma adição a plataforma de software é definida através de modelos UML focados em uma plataforma. Para isso são utilizados *OpaqueBehaviors* da UML. A UML declara *OpaqueBehavior* como sendo um comportamento com semântica específica para uma determinada plataforma

(OMG, 2009b). Neste caso, todos os elementos, assim como seus atributos e seus comportamentos devem ser modelados, sendo eles transformados em código juntamente com a transformação em código do restante do sistema.

Para ambas as abordagens, o presente trabalho prega a definição de um modelo adicional para este fim, que deve ser entrelaçado com o modelo da plataforma para se obter o modelo PDM – Software. Tal entrelaçamento é realizado pela mesma transformação para entrelaçamento de modelos definida na seção referente ao PIM, pois as regras definidas são aplicáveis.

O presente trabalho não tem como objetivo avaliar as vantagens e desvantagens de cada abordagem, mas preferiu modelar o quadro de trabalho de comunicação entre objetos usando a segunda abordagem. Tal quadro de trabalho é detalhado na seção seguinte.

4.3.2.1. Mecanismo de comunicação entre objetos

Um ponto fundamental definido na PDM Software é o mecanismo de comunicação entre objetos, o presente trabalho define um modelo especialmente para esta finalidade. O objetivo desta separação é permitir que uma mesma plataforma de software possa chavear entre mecanismos de comunicação, entretanto este chaveamento deve ser realizado juntamente com alterações na transformação de PIM para PSM.

Este modelo segue a abordagem anteriormente explorada, onde todos os elementos são modelados usando-se UML. Tal modelo contém: classes, interfaces, atributos, operações e *OpaqueBehaviors* (contendo código na linguagem selecionada). Cada operação (UML *Operation*) deve possuir o atributo *Method* preenchido com um *OpaqueBehavior*. É importante ainda definir os elementos *ElementImport* em cada elemento para garantir a

conformidade do modelo com ferramentas de validação e também na tradução adequada para a plataforma alvo.

Apesar deste modelo, para a definição do mecanismo de comunicação entre objetos, ser separado do restante da PDM – Software, ele exibe um alto grau de dependência das camadas mais baixas da plataforma. Por exemplo, baseando-se na plataforma Java, um método chamado *send* precisa receber pelo menos um parâmetro com a mensagem a ser exibida, uma das alternativas para definir este parâmetro é utilizar a classe *java.lang.Object*. Ou seja, para modelar o mecanismo de comunicação entre objetos é preciso usar os elementos anteriormente definidos na PDM – Software. Este é um caso claro das pontes explícitas (MELLOR e BALCER, 2002), onde um modelo depende diretamente do outro, ou melhor, existe uma ligação direta entre o modelo PDM – Software e o modelo que define o mecanismo de comunicação entre objetos. O que é bem diferente das pontes implícitas, onde os terminadores impedem a dependência direta.

Considerando um modelo fUML no nível PIM, existem três cruciais questões que um modelo de mecanismo para comunicação entre objetos em sistemas de tempo real deve responder:

1. Como ele define a tradução da ação *ReadExtent* (usada como ferramenta para desacoplamento)?
2. Como ele trata a comunicação entre objetos ativos e passivos?
3. Se o código de tal mecanismo é aderente a um sistema de tempo real?

Como citado na seção referente à modelagem usando fUML, a ação *ReadExtent* não é tranquilamente traduzida para código de linguagens usuais de programação, pois assume a existência de um índice com todos os objetos instanciados. Tal índice é naturalmente disponibilizado por um sistema de gerenciamento de banco de dados. Em sistemas de tempo real alternativas

devem ser avaliadas para a tradução desta ação, por exemplo, no momento da transformação as instancias podem ser localizadas e um relacionamento pode ser estabelecido entre o objeto que está executando a ação *ReadExtent* e as instancias localizadas.

O presente trabalho optou por modelar na avaliação de aplicabilidade um mecanismo que não acoplasse fortemente os objetos envolvidos, usando a alternativa chamada por Gomaa (2000) de “comunicação fracamente acoplada baseada em mensagens”, *Loosely Coupled (Asynchronous) Message Communication*, também chamada de comunicação assíncrona. Nela o produtor envia uma mensagem para o consumidor e continua seu processamento, uma fila FIFO (*First-in-First-Out*) é introduzida como ferramenta de desacoplamento entre o produtor e o consumidor. Ela tem a vantagem de manter naturalmente um índice das instancias que aguardam o recebimento de sinais (essencialmente assíncronos). Isto vale apenas para objetos ativos, para a comunicação entre objetos passivos o uso de chamadas diretas a operações é, não só aceitável como, recomendada, desde que os requisitos de sincronização sejam respeitados.

Assim entre as duas grandes classes de soluções para a construção de um programa concorrente (FARINES et al., 2000), sendo elas troca de mensagens e variáveis compartilhadas, o presente trabalho optou pela troca de mensagens. Farines et al. (2000) ainda define que a troca de mensagens é baseada em duas operações simples: enviar mensagem e receber mensagem, e que tanto a comunicação como a sincronização são realizadas através das mesmas operações. Já as soluções baseadas em variáveis compartilhadas usam intensamente algum mecanismo de sincronização.

O mecanismo de comunicação entre objetos, assim como a semântica de simultaneidade, são fatores cruciais para definir a quantidade de bloqueio (*blocking*) em um sistema de tempo real. Briand e Roy (1999) destacam que

sempre existirão várias alternativas de projeto para uma aplicação, e que a quantidade de bloqueio é um dos fatores que definem se um projeto é bom ou mau. Segundo Briand e Roy (1999) a redução do bloqueio é similar a redução na dependência entre módulos, que sempre é benéfica.

Sobre o código ser aderente ou não a um sistema de tempo real, Kopetz (1997) define três regras para projeto:

- Ausência de laços sem um limite predeterminado;
- Ausência de chamadas recursivas;
- Ausência de estruturas dinâmicas de dados.

Estas três regras foram seguidas na implementação usada para fins de protótipo, o que não quer dizer, de modo algum, que tal protótipo pode ser usado em uma aplicação real.

Para encerrar é preciso destacar que mais um pacote do MARTE foi utilizado, é o SRM. Mais precisamente, o estereótipo *MessageComResource*, cuja definição é: são artefatos para comunicar mensagens entre recursos concorrentes (OMG, 2009a). Tal estereótipo anota a classe responsável pela implementação do mecanismo de comunicação entre objetos.

4.3.3. Conclusão e ferramentas utilizadas

Antes de passar para o detalhamento da PSM é preciso indicar quais ferramentas foram utilizadas para a modelagem da PDM. Todos os modelos PDM foram construídos utilizando-se a ferramenta TOPCASED 3.3 (TOPCASED, 2010). Todos estes modelos são entrelaçados usando-se a mesma transformação definida para entrelaçar modelos no nível PIM gerando um único modelo PDM. Tal transformação foi executada usando-se a ferramenta Eclipse Galileo Modeling (ECLIPSE FOUNDATION, 2010b).

4.4. PSM

Neste ponto existe um modelo PIM, composto pelo entrelaçamento de diversos PIMs, através de uma transformação MOF QVTO (OMG, 2008a). Existe também um modelo PDM, composto pelo entrelaçamento dos modelos PDM, usando-se a mesma transformação MOF QVTO. Dado que o presente trabalho selecionou o mapeamento explícito, o objetivo desta seção é detalhar como um modelo PSM é gerado a partir destes dois modelos (PIM e PDM).

A seguir serão abordados dois tópicos:

- Transformação MOF QVTO – responsável pela geração do modelo PSM, definindo toda a **estrutura** do sistema e disparando a *blackbox library* para geração de comportamento;
- *Blackbox library* – responsável pela geração do **comportamento** para a plataforma alvo selecionada através da análise das atividades definidas usando-se fUML.

A próxima seção detalha a transformação MOF QVTO, e por fim a *blackbox library* é explorada.

4.4.1. Transformação MOF QVTO

Antes de detalhar a abordagem usada para fins do protótipo no presente trabalho, é importante lembrar que a transformação de PIM para PSM é extremamente dependente da PDM, principalmente da parte da PDM chamada anteriormente de PDM – Software. Pois ela é responsável por traduzir conceitos independentes de plataforma, usados no PIM, para conceitos dependentes de plataforma. Por exemplo, esta transformação é responsável por transformar classes ativas em classes que estendem a classe *java.lang.Thread*, considerando a plataforma Java. Tal transformação só pode ser realizada com sucesso se pelo menos dois pontos forem satisfeitos:

- Existência da classe *java.lang.Thread* modelada na PDM;

- Regra de transformação baseada no meta-modelo UML codificada na transformação, indicando que cada classe ativa deve ser traduzida em outra classe que estende a classe *java.lang.Thread*.

Até aqui fica claro que, uma alteração radical na PDM, principalmente na PDM – Software leva a uma revisão completa da transformação de PIM para PSM, ou seja, a transformação complementa a PDM formando o que pode ser chamado de uma plataforma. Isto é o que Kennedy Carter (2002) chama de uma abordagem centrada em arquitetura, onde um conjunto PDM + PSM é construído para cada arquitetura e plataforma. Para explorar bem este ponto, se o mesmo modelo PIM da avaliação de aplicabilidade desenvolvida no presente trabalho tivesse que ser traduzido para uma plataforma de software distinta, por exemplo, linguagem C, usando o mesmo hardware, todo o modelo PDM – Software teria que ser construído a partir do zero assim como a transformação de PIM para PSM. Mas uma vez, que este modelo PDM e a nova transformação estivessem prontos, eles poderiam ser reutilizados para sistemas com arquitetura e plataforma igual.

É importante destacar que se um erro for localizado em um PDM ou em uma transformação (para PSM), tal erro estará replicado em todos os sistemas gerados pela plataforma (PDM + Transformações) em questão. Conseqüentemente, se um erro for corrigido em uma plataforma todos os sistemas podem ser corrigidos automaticamente através da re-geração do código.

É possível notar como o reuso de conhecimento tem forte ênfase na MDA, uma vez que foi definida uma forma para traduzir modelos PIM em modelos PSM tal conhecimento pode ser reusado de forma intensa, e se um eventual erro no processo de transformação for encontrado o ponto a ser mantido é a transformação e não um processo ou uma lista de *best practices*.

No presente trabalho, tal transformação é codificada usando-se a especificação MOF QVTO focada na plataforma Java de software e em um único dispositivo de processamento.

Optou-se por entrelaçar (usando a mesma transformação explorada para entrelaçamento de modelos) o modelo PIM com o PDM, gerando um terceiro modelo que simplesmente contem todos os elementos do PIM e do PDM. Este modelo é o principal parâmetro de entrada para MOF QVTO. É possível visualizar a declaração da transformação, assim como seus parâmetros na Figura 4.16.

```
import m2m.qvt.oml.PDMJavaLib;

modeltype UML      uses uml('http://www.eclipse.org/uml2/3.0.0/UML');
modeltype EMF      uses ecore('http://www.eclipse.org/emf/2002/Ecore');

transformation PIM2PSMJava( in pimpdm_woven:UML, in pdm:UML, in marte:UML, out psm:UML)
          access PDMJava;

-- define default package for classes that came from PIM
configuration property defaultPackage : String;

-- main
main() {
```

Figura 4.16 – Transformação MOF QVTO PIM para PSM.

Destacam-se na Figura 4.16 os seguintes pontos:

- Meta-modelos, descreve os meta-modelos usados na transformação:
  - UML – é o meta-modelo na UML;
  - EMF – é o meta-modelo do *Eclipse Modeling Framework* (EMF), ele é usado como base para a definição do meta-modelo da UML no eclipse, sendo utilizado na transformação para manipular estereótipos;
- Parâmetros de entrada:
  - pimpdm_woven – usado para informar o principal parâmetro de entrada, sendo ele o resultado do entrelaçamento entre o modelo PIM e o PDM;

    - pdm – usado para determinar se um dado elemento pertence ao PDM ou ao PIM, está informação é importante, pois elementos advindos do PDM já estão prontos para o PSM, enquanto que os provenientes do PIM precisam ser transformados;
    - marte – trata-se do UML *profile* MARTE completo, com todos os seus pacotes. Ele é recebido como parâmetro de entrada para que seja usado para anotar automaticamente classes a serem geradas no PSM;
- Parâmetros de saída:
    - psm – trata-se do modelo PSM gerado pela transformação;
- Acesso a *blackbox library*:
    - PDMJava – trata-se da *blackbox library* responsável pela geração de comportamento do modelo PSM.

Existe ainda um parâmetro de configuração, chamado *defaultPackage*, ele é usado para adicionar um prefixo ao pacote de elementos provenientes do PIM, pois os mesmos não devem seguir a convenção de pacotes Java ou de implementação de uma determinada plataforma.

Dado os parâmetros de entrada é preciso esclarecer as regras de transformação, baseando-se no meta-modelo UML, a Tabela 4.2 lista de forma resumida as regras codificadas no protótipo.

Tabela 4.2 – Resumo da regras utilizadas PIM para PSM

Origem (meta-modelo UML)	**Destino (meta-modelo UML)**	**Restrição**
Profile Application	*Profile Application*	
Stereotype Application	*Stereotype Application*	
Package	*Package*	Se possuir elementos filhos e nome diferente de *ClockLibrary*
Activity	*Class* <<*MemoryPartition*>>	Se for uma atividade na raiz do modelo e não

		começar com “*test*”
Class	*Class*	Se *isActive=false* e não possuir o estereótipo *DeviceResource*
Class *<<DeviceResource>>*	*Class* *<<DeviceBroker>>*	Se *isActive=false* e possuir o estereótipo *DeviceResource*
Class	*Class extends java.lang.Thread* *<<SwSchedulableResource>>* *<<EntryPoint>>*	Se *isActive=true*
Interface	*Interface*	
PrimitiveType	*PrimitiveType*	
Signal	*Class implements java.io.Serializable*	
Property	*Property*	
Operation	*Operation* *OpaqueBehavior*	Se vier do PDM
Operation	*Operation* *OpaqueBehavior (generated)*	Se vier do PIM
Parameter	*Parameter*	
Comment	*Comment*	
UML::PrimitiveType::Integer	*double (Java) / 1000000d*	
UML::PrimitiveType::UnlimitedNatural	*int (Java)*	
UML::PrimitiveType::Boolean	*boolean (Java)*	
UML::PrimitiveType::String	*java.lang.String (Java)*	
SignalEvent	*-*	
ElementImport	*ElementImport*	
-	*ElementImport*	
-	*Profile Application* *<<SW_Concurrency>>* *<<SW_Broker>>*	
-	*Abstraction*	

Três destas regras serão exploradas agora em mais detalhes de modo a prover ao leitor um detalhamento mais profundo do funcionamento da abordagem proposta.

Sobre a transformação de tipos básicos UML para tipos básicos Java, merece destaque a transformação dos elementos do tipo *UML::PrimitiveType::Integer*. Como proposto na seção referente à modelagem através de fUML todas as constantes inteiras dos modelos PIM foram multiplicadas por 1000000 para permitir que os cálculos exibissem um mínimo de precisão, entretanto no momento de traduzir estes elementos para o PSM, como o tipo básico Java prove uma boa precisão e o modelo irá descrever o código final, está simplificação de modelagem no nível PIM precisa ser removida do PSM. Assim existe uma regra na transformação, que pode ser descrita como: todo o elemento do tipo *UML::PrimitiveType::Integer* deve ser transformado em um elemento do tipo Java *double*, nesta tradução caso existam constantes tais constantes devem ser divididas por 1000000d.

Sobre a transformação de classes ativas no PIM para classes que estendem *java.lang.Thread* no PSM, a Figura 4.17 mostra um trecho da transformação focada nesta regra.

```
------------------------------------------------------------
-- Active Classes
mapping Class::toActiveClass() : UML::Class
when { self.oclIsTypeOf(Class) and ( not self.isFromPDM() and self.isActive)}
{
population{
        var start:= object OpaqueBehavior {
                name:="start";
                body:="super.start();";
                };
        var startOp:= object Operation {
                name:="start";
                method := start;
                };
        result.ownedBehavior += start;
        result.ownedOperation += startOp;

        var run:= object OpaqueBehavior {
                name:="run";
                language := "Java";
                body := self.classifierBehavior.oclAsType(Activity).getBodyToActivity(); //call blackbox library
                };
        var runOp:= object Operation {
                name:="run";
                method := run;
                };
        result.ownedBehavior += run;
        result.ownedOperation += runOp;

        //extends thread
        object Generalization {
                general := pdm.objectsOfType(Class)->selectOne(qualifiedName.substringAfter("::") = "java::lang::Thread");
                specific := result;
                };
```

Figura 4.17 – MOF QVTO Transformação classes ativas.

Inicialmente é preciso notar que o mapeamento é iniciado a partir de uma classe e que gera como resultado outra classe, além disso, a cláusula *when* determina que apenas classes ativas não provenientes do PDM sofrerão esta transformação.

É possível notar que primeiro o *OpaqueBehavior* para a *Operation* start é criado, contendo já a linha de código Java. Em seguida, a *Operation* start é criada e associada a ela o *OpaqueBehavior* criado anteriormente. Na seqüencia comandos similares são executados para a *Operation run*, a única diferença é que neste caso o código Java é gerado através de uma chamada a *blackbox library*. Esta *blackbox library* será responsável por avaliar a atividade recebida e traduzi-la em código Java. Por último é criada uma *Generalization* entre a classe sendo criada, *result*, e a classe *java::lang::Thread*. É preciso dar ênfase especial ao uso intenso do meta-modelo da UML, e que tal parte da transformação esta essencialmente focada em gerar a estrutura da classe, sendo seu comportamento gerado pela *blackbox library*.

Fowler (2005) destaca que um diagrama de estados pode ser implementado de três maneiras principais: comando *switch* aninhado, o padrão *State* e tabelas de estado. Entretanto o presente trabalho preferiu simplificar a transformação implementando as atividades (máquinas de estado) em um fluxo linear de programação.

Ainda sobre esta regra cada classe gerada precisa ser automaticamente anotada com o estereótipo *SwSchedulableResource* com os atributos prazo e período preenchidos adequadamente mediante as informações obtidas nas NFPs definidas no PIM.

A última regra a ser explorada em detalhes e a regra referente a atividades de inicialização, elas foram definidas no PIM basicamente para a criação de objetos e inicialização de comportamento dos mesmos. No momento de

transformação, as atividades que forem encontradas na raiz do modelo são traduzidas em *OpaqueBehaviors* pela *blackbox library*, então *Operations* são criadas para cada uma delas, em seguida uma classe de inicialização é criada, sempre chamada de *GeneratedForAloneactivities*, e por fim as operações previamente criadas são associadas ao *main* desta classe. Após isso, é preciso aplicar automaticamente o estereótipo *MemoryPartition* do SRM na classe *GeneratedForAloneactivities*.

Um ponto importante na modelagem de sistemas é permitir a rastreabilidade entre os modelos, a transformação prototipada no presente trabalho cria objetos UML chamados *Abstraction* documentando o relacionamento entre um elemento no PSM e um elemento no PIM.

O último ponto que merece destaque é o calculo automático dos elementos UML *ElementImports*, eles são essenciais para que a geração de código seja simples, pois garantem que todos os relacionamentos necessários para a compilação de uma classe estão declarados no modelo PSM. Eles são calculados dinamicamente pela transformação através da avaliação de classes e métodos referenciados por determinado elemento.

Após a aplicação de tais regras, existe um trabalho significativo que é corrigir as referências, removendo referências dos modelos de entrada e apontando para o modelo de saída. Por exemplo, a classe *java.lang.Thread* estará no modelo PDM e também no PSM, é preciso garantir que todas as referências a tal classe a partir do PSM apontem para o elemento no modelo PSM.

A seguir a *blackbox library* será explorada em mais detalhes.

4.4.2. Blackbox library

Como discutido anteriormente existem alguns tipos de transformação que dificilmente são implementadas usando-se o formalismo da especificação MOF QVT, para isso tal especificação define o tipo de mapeamento chamado *blackbox* (OMG, 2008a).

A Figura 4.17 mostra como é disparada uma *blackbox library* responsável pela tradução de atividades em código Java. A Figura 4.18 exibe justamente o método invocado quando a linha marcada como "*//call blackbox library*" da Figura 4.17 é executada.

```
public class PDMJava {

	public PDMJava() {
		super();
	}

	/**
	 * Blackbox library for code generation, using opaque behavior.
	 *
	 * @param activity
	 * @return
	 */
	@Operation(kind = Kind.OPERATION, contextual = true)
	public String getBodyToActivity(Activity activity) {
		EList<ActivityNode> nodes = activity.getNodes();

		ActivityNode initialNode = null;
		for (ActivityNode activityNode : nodes) {
			if (activityNode instanceof InitialNode) {
				initialNode = ((InitialNode) activityNode);
				break;
			}
		}

		ActivityDiagramCodeGenerator codeGenerator = new ActivityDiagramCodeGenerator(
				this);

		return codeGenerator.getCode(initialNode);
	}
```

Figura 4.18 – MOF QVTO *Blackbox library.*

No presente protótipo uma classe chamada *ActivityDiagramCodeGenerator* é invocada passando-se o nó inicial da atividade, dentro desta classe existe um método para cada nó selecionado para prototipação. Dentro cada método o respectivo código é gerado.

Um ponto mandatório para a avaliação de atividades e geração de código é a semântica definida na UML para execução de ações. Seguindo a UML (OMG,

2008b) uma ação só pode começar a ser executada quando todos os fluxos de controle possuírem *tokens* e todos os *InputPins* possuírem *tokens* com objetos.

Para exemplificar, a Figura 4.19 apresenta o método responsável pela geração de código para a ação *CreateObjectAtion*.

```
/**
 * @param node
 */
private void generateCode(CreateObjectAction node) {
        builder.append("\r\n" + pdmJava.getPSMClassName(node.getClassifier())
                        + " " + getJavaVariableName(node.getResult()) + " = new "
                        + pdmJava.getPSMClassName(node.getClassifier()) + "();");

        // registering variable
        variableNameType.put(getJavaVariableName(node.getResult()), pdmJava
                        .getPSMClassName(node.getClassifier()));
}
```

Figura 4.19 – Método que gera código para *CreateObjectAtion.*

O resultado do método apresentado na Figura 4.19 é a seguinte linha de código Java:

Class variable = new Class();

Outra característica que vale a pena ser destacada é que, o tradutor de atividade para código Java gera uma linha para cada ação. Isto gera um código final não otimizado, mas isso é aceitável para fins do protótipo. A seguir são apresentadas as conclusões e as ferramentas utilizadas.

4.4.3. Conclusões e ferramentas utilizadas

A transformação foi criada e executada usando-se a ferramenta Eclipse Galileo Modeling (ECLIPSE FOUNDATION, 2010b). A *blackbox library*, na verdade um *plugin* do Eclipse, também foi criada e executada usando-se a ferramenta Eclipse Galileo Modeling (ECLIPSE FOUNDATION, 2010b).

Após a execução da transformação existe um PSM que descreve, já em termos da plataforma alvo, todo o sistema. Neste ponto um especialista em software deve complementar este modelo com informações adicionais, um exemplo importante é a definição do tempo de ativação de uma classe anotada automaticamente com o estereótipo *SwSchedulableResource*. Esta informação é crucial para a avaliação de escalonabilidade, mas não é definida como uma NFP no PIM, pois depende de vários fatores, entre eles destacam-se: código gerado, a plataforma de software e de hardware. O presente trabalho sugere o uso de informações históricas ou de avaliações de pequenas partes de código na plataforma alvo.

4.5. Código

Para gerar o código para a plataforma de software selecionada, na presente proposta Java, uma transformação MOF M2T (OMG, 2008b) é aplicada ao PSM. Cabe aqui destacar, que esta geração de código é simplesmente uma "serialização" do modelo, já que o modelo contém toda a estrutura e comportamento já contemplando a plataforma alvo.

O código automaticamente gerado contempla toda a regra de negócio e adequações a plataforma selecionada assim ele não precisa ser modificado, entretanto podem existir casos onde a customização do código seja necessária, para isso a especificação MOF M2T (OMG, 2008b) prove meios para adição de código sem comprometer as subseqüentes execuções de transformações ou correr o risco de perda do código inserido, são os chamados blocos protegidos (*protected*).

Para ficar mais claro como funciona esta transformação para código, a seguir será explorada a geração de um método em Java a partir de uma operação UML (*Operation*) definida no modelo PSM. A Figura 4.20 detalha a geração de um método a partir de uma operação no PSM.

```
[comment]
        operations - opaque behaviors
[/comment]
[template private operationBody(o : Operation)]
        /**
        *
    [for (cmt : Comment | o.ownedComment)]
        * [cmt.body/]
    [/for]
        [for (p : Parameter | o.parameters())]
        * @param [p.name/] [for (cmt : Comment | p.ownedComment)] [cmt.body/][/for]
        [/for]
        [if (not o.returnType().oclIsUndefined())]
        * @return
        [/if]
    * @mygenerated "sourceid:[o.eResource()/]#[o.eClass().getClassifierID()/]"
        */
    public [if (o.isSynchronized())]synchronized [/if][if (o.isLeaf)]final [/if] [if
(o.isStatic)]static [/if][o.returnTypeOperation()/] [o.name/]([o.getInParameter()/]) {
        // [protected ('for operation '.concat(o.name))]
        // NOT RECOMMENDED - CAN BE implemented
        // [/protected]
            [for (bodyLine : String | o.method.oclAsType(OpaqueBehavior).body)]
                [bodyLine/]
        [/for]
    }
[/template]
```

Figura 4.20 – MOF M2T PSM para código – Operações.

Para cada operação são gerados:

- Comentários, usando-se as informações do PSM, seguindo o formato *javadoc*;
- Um comentário especial, *mygenerated*, dedicado a rastreabilidade, usado para indicar no código gerado qual foi o elemento do PSM que originou tal método. A especificação MOF M2T (OMG, 2008b) define um bloco para rastreabilidade, chamado *trace*, entretanto o *plugin* utilizado (OBEO, 2010) não implementa tal bloco. Por convenção este comentário deveria ter o nome *generated*, entretanto o uso deste nome causa erros no *plugin*, justamente por ser um nome reservado a rastreabilidade;
- Assinatura do método, avaliando-se várias propriedades da operação, se ela é sincronizada, se ela é folha, se ela é estática, se possui parâmetros de retorno, se possui parâmetros de entrada;

- Bloco protegido, demarcando partes do código que podem ser alteradas diretamente, sem risco de perda em uma eventual nova transformação de PSM para código;
- Corpo do método, escrito a partir do *OpaqueBehavior* associado ao atributo *Method* da operação.

Como o presente trabalho selecionou o mapeamento explícito, aquele onde um modelo intermediário, contendo apenas os conceitos disponibilizados pela plataforma alvo (MRAIDHA, 2006), é introduzido, a transformação do modelo PSM para código é bem simples, pois sua única responsabilidade é traduzir o modelo PSM em um conjunto de arquivos seguindo o formato definido pela linguagem destino, no presente trabalho Java.

Para reforçar o quanto a geração de código é simplificada pela existência do PSM, um ponto complexo na geração automática de código em Java é a determinação das clausulas *import*, na presente proposta eles são geradas através dos elementos UML *ElementImports* já disponíveis em cada classe da PSM.

4.5.1. Código Java

A Figura 4.21 mostra um trecho de código gerado para uma operação do modelo PSM.

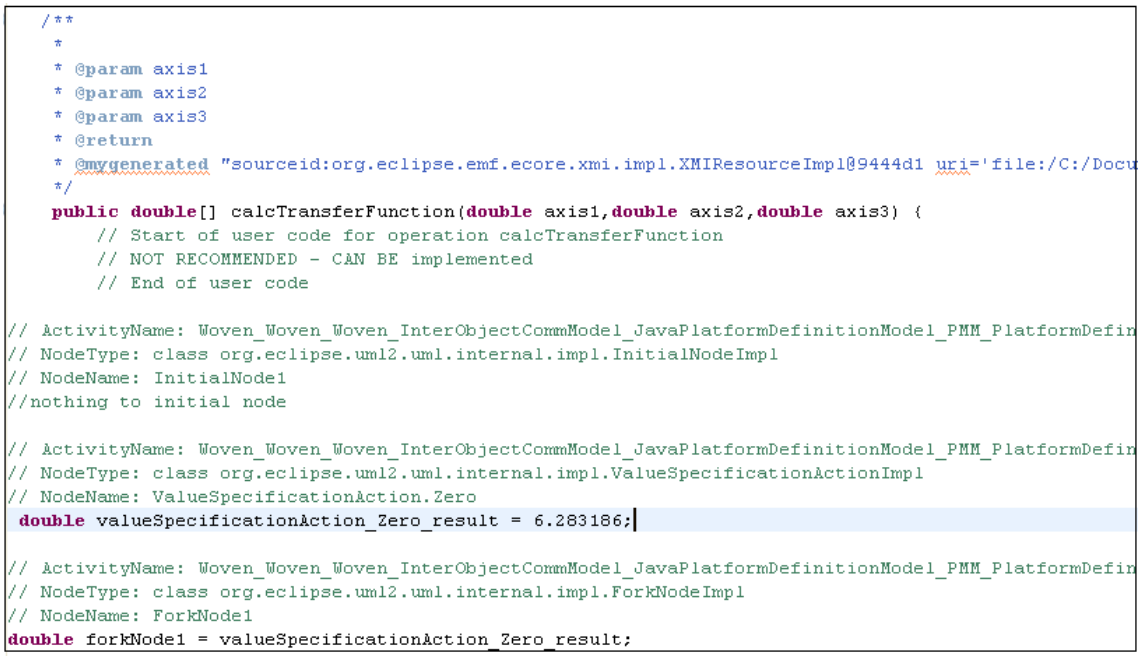

Figura 4.21 – MOF M2T PSM para código – Protótipo Java.

Destaca-se a definição do tipo de retorno e dos tipos dos parâmetros de entrada, assim como do bloco para alteração do código e também do código de comportamento gerado automaticamente. É possível visualizar na Figura 4.21 que o código gerado na PSM para a operação, armazenado em um *OpaqueBehavior*, possui detalhes de que código foi gerado para que ação.

4.6. AADL

Uma vez que existe um modelo PSM, e este foi complementado com as informações sobre o tempo de ativação de cada elemento que contem o estereótipo *SwSchedulableResource*, é possível realizar a análise de escalonabilidade. Para isso, um modelo AADL é crucial.

O presente trabalho optou por transformar o PSM em um modelo AADL, mesmo que parcial, mas que permitisse a avaliação de escalonabilidade em qualquer ferramenta capaz de interpretar modelos AADL.

Para gerar um modelo AADL uma transformação MOF M2T (OMG, 2008b) é aplicada ao PSM. Esta transformação é simples, já que o modelo PSM contém toda a informação necessária. Para a geração do AADL destacam-se período, prazo e tempo de ativação para cada tarefa, assim como objetos compartilhados entre as mesmas. Para gerar o modelo AADL as regras resumidas na Tabela 4.3 são utilizadas.

Tabela 4.3 – Resumo da regras utilizadas PSM para AADL

Origem (meta-modelo UML)	**Destino (meta-modelo AADL) formato texto**	**Restrição**
Model	*System*	
Class <<HwProcessor>>	*Processor*	
Class <<Scheduler>>	*Processor.SchedulingProtocol*	
Class <<MemoryPartition>>	*Process*	
Class <<SwSchedulableResource>>	*Thread*	
Class	*Data*	Se *isActive=false*

Para esclarecer como o modelo AADL é gerado, a seguir são exploradas duas das regras aplicadas.

Para gerar o sistema (*system*), são disparadas duas buscas na PSM, a primeira para localizar os processadores, ou seja, são localizadas todas as classes que possuem o estereótipo *hwProcessor*. Em seguida, são localizadas todas as classes que possuem o estereótipo *memoryPartition*. Todos os elementos localizados são definidos como subcomponentes do sistema, e então a seguinte verificação é realizada, o atributo *memorySpaces* do estereótipo *memoryPartition* aponta para um mesmo elemento do atributo *ownedHW* do estereótipo *hwProcessor*, se sim é declarada então uma relação entre o processador (*hwProcessor*) e o processo (*memoryPartition*).

A Figura 4.22 ilustra o modelo AADL gerado na avaliação de aplicabilidade realizada no presente trabalho.

```
-- AADL
-- transformed from PSM

system PSM
end PSM;
system implementation PSM.Impl
        subcomponents
                instancied_CPU : processor CPU.Impl;
                instancied_GeneratedForAloneActivitiesImpl : process GeneratedForAloneActivitiesImpl.Impl;
        properties
                Actual_Processor_Binding => reference instancied_CPU applies to instancied_GeneratedForAloneActivitiesImpl;
end PSM.Impl;


processor CPU
        features
                bus_connected : requires bus access ;
end CPU;
processor implementation CPU.Impl
        properties
                Scheduling_Protocol => RATE_MONOTONIC_PROTOCOL;
end CPU.Impl;


process GeneratedForAloneActivitiesImpl
end GeneratedForAloneActivitiesImpl;
process implementation GeneratedForAloneActivitiesImpl.Impl
        subcomponents
                instancied_ControlReceiverImpl : thread ControlReceiverImpl.Impl;
                instancied_ControlSensorRTEntityImpl : thread ControlSensorRTEntityImpl.Impl;
                instancied_GyroscopeSensorDevice : data GyroscopeSensorDevice.Impl;
                instancied_MessageQueueConnector : data MessageQueueConnector.Impl;
                instancied_GyroscopeTransferFunction : data GyroscopeTransferFunction.Impl;

        connections
                data access instancied_MessageQueueConnector -> instancied_ControlReceiverImpl.MessageQueueConnector_featur
                data access instancied_MessageQueueConnector -> instancied_ControlSensorRTEntityImpl.MessageQueueConnector_
                data access instancied_GyroscopeTransferFunction ->
RTEntityImpl.GyroscopeTransferFunction_features;
                data access instancied_GyroscopeSensorDevice -> instancied_ControlSensorRTEntityImpl.GyroscopeSensorDevice_

end GeneratedForAloneActivitiesImpl.Impl;
```

Figura 4.22 – Trecho do modelo AADL – Sistema.

Para gerar uma linha de execução (*thread*), uma avaliação de uma classe anotada com o estereótipo *SwSchedulableResource* é executada. Para se descobrir os elementos referenciados, os elementos UML chamados *ElementImports* são avaliados. Em seguida, as propriedades do estereótipo *SwSchedulableResource* são usadas para determinar o protocolo de despacho, o tempo de execução (tempo de ativação no MARTE), o prazo, o período e também a incerteza de liberação (*release jitter*).

Sobre a incerteza de liberação, é comum que algumas tarefas só não possam ser executadas em ordem arbitrária, mas sim, em ordens previamente definidas, o que determina o surgimento de relações de precedência (FARINES et al., 2000). Uma técnica para implementar relações de precedência é o uso de mensagens. O presente trabalho está fortemente embasado na troca de mensagens entre tarefas, e neste modelo uma conseqüência direta é a

existência de incertezas de liberação (FARINES et al., 2000) (DOUGLAS, 1999).

A Figura 4.23 ilustra uma tarefa no modelo AADL gerado na avaliação de aplicabilidade realizada no presente trabalho.

```
thread ControlReceiverImpl
        features
                MessageQueueConnector_features : requires data access MessageQueueConnector.Impl;
end ControlReceiverImpl;

thread implementation ControlReceiverImpl.Impl
        properties
                Dispatch_Protocol => Periodic;
                Compute_Execution_Time =>  15 ms ..  15 ms;
                Deadline  =>  100 ms;
                Period =>  100 ms;
                Cheddar_Properties::Dispatch_Jitter =>   0 ms;
end ControlReceiverImpl.Impl;


data MessageQueueConnector
end MessageQueueConnector;
data implementation MessageQueueConnector.Impl


end MessageQueueConnector.Impl;
```

Figura 4.23 – Trecho do modelo AADL – Tarefa.

Destaca-se o acesso ao mecanismo de comunicação entre objetos (*MessageQueueConnector*), e as propriedades da tarefa (protocolo de despacho, período, tempo para ativação, prazo e incerteza de liberação).

4.6.1. Conclusões e ferramentas utilizadas

Dado que existe um modelo AADL, é possível utilizar uma ferramenta para realizar a análise de escalonanilidade.

Conforme descrito anteriormente o presente trabalho selecionou a ferramenta Cheddar (LISYC, 2010) para realizar a análise de escalonabilidade. A seguir são exibidas duas figuras:

- A Figura 4.24 ilustra um teste de escalonabilidade realizado na avaliação de aplicabilidade – nela destacam-se dois resultados:

- É possível antecipar o percentual de uso do processador, um requisito usual para sistemas de tempo real (BRIAND e ROY, 1999);
- É garantido matematicamente que o conjunto de tarefas é escalonável;

- A Figura 4.25 ilustra uma simulação de execução voltada para análise de escalonabilidade.

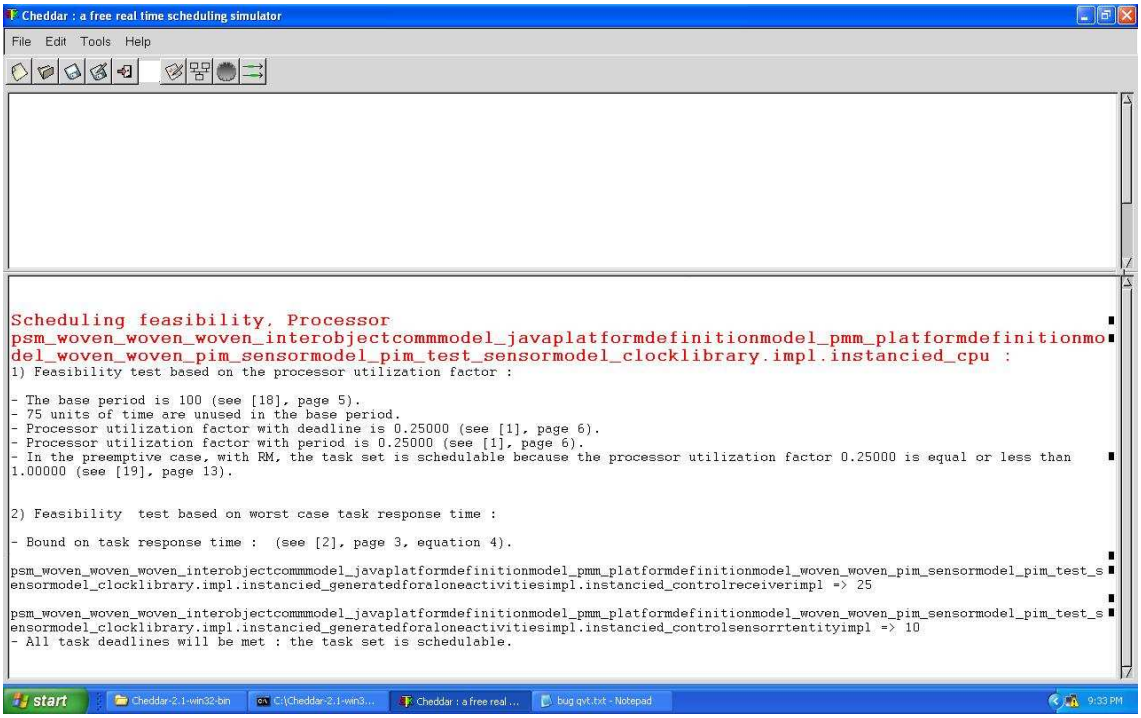

Figura 4.24 – Análise de escalonabilidade, teste de escalonabilidade.

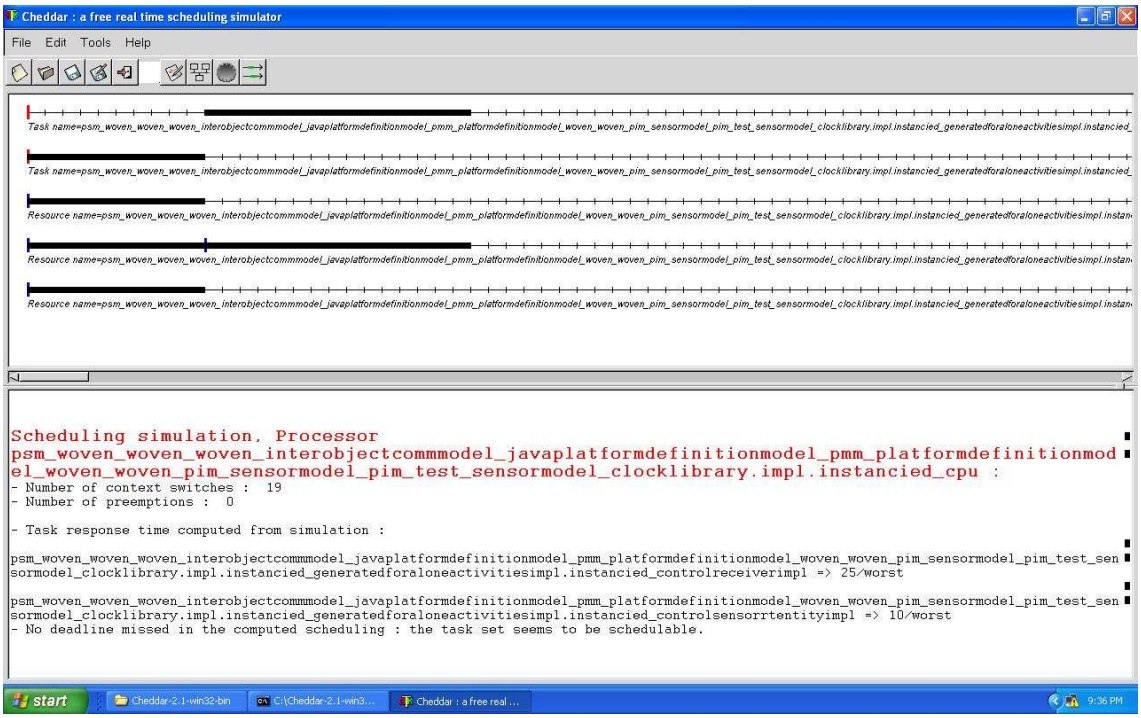

Figura 4.25 – Análise de escalonabilidade, através de simulação.

Com isso está concluído o trabalho, foram detalhados desde a forma de modelagem do PIM, passando pelas transformações até a geração de código e de um modelo AADL, e ainda sobre o modelo AADL foram realizadas análises de escalonabilidade. O próximo capítulo apresenta os resultados do trabalho.

5 RESULTADOS

Este capítulo descreve os resultados para cada objetivo específico definido, assim como as fragilidades e limitações do presente trabalho.

5.1. Resultados para cada objetivo específico

1. Definir o que deve ser modelado e em que nível (PIM ou PSM);

A abordagem proposta, baseada na MDA, para a confecção de modelos PIM permite que subsistemas e sistemas sejam modelados usando ações e atividades, que tais modelos possuam tanto propriedades funcionais quanto não funcionais e, ainda, que tais modelos sejam automaticamente verificáveis. No decorrer do presente trabalho foi possível perceber o reuso de conhecimento expresso nos modelos, no lugar do tradicional reuso de código.

A abordagem proposta, baseada na MDA, para a geração automática de modelos PSM a partir do PIM e de uma PDM permite que características de hardware e software sejam consideradas na geração do PSM. Além disso, o PSM gerado permite a geração automática de um modelo AADL, usado para avaliação de escalonabilidade.

2. Definir o que usar de cada especificação selecionada, sendo elas: fUML, MARTE, MOF, MOF QVT e MOF M2T;

Durante os capítulos anteriores foi apresentada uma descrição detalhada do que foi usado de cada especificação selecionada inicialmente para o presente trabalho.

3. Criar protótipos das transformações:

a. PIM para o PSM;

Um protótipo para esta transformação, a principal na MDA, foi desenvolvido usando-se a especificação MOF QVTO e o conceito de *blackbox library*. Ela é desenhada para a plataforma Java e depende dos modelos PDM – Software – Java e PDM – Software – Java – Comunicação entre objetos.

b. PSM para AADL;

Um protótipo para esta transformação foi desenvolvido usando-se a especificação MOF M2T.

c. PSM para código Java.

Um protótipo para esta transformação foi desenvolvido usando-se a especificação MOF M2T.

4. Usar as definições anteriores na modelagem de um PIM para a avaliação de aplicabilidade considerando parte do subsistema de controle de atitude da PMM, seguindo a modelagem de controle apresentada em Moreira (2006);

O apêndice A detalha o que foi modelado partindo-se do trabalho de (Moreira, 2006), assim como as alterações introduzidas. Neste apêndice são exibidos ainda os modelos de teste, o modelo AADL e parte do código automaticamente gerado.

5. Testar funcionalmente, de modo parcial através de um protótipo, o modelo PIM;

O capítulo anterior apresentou a abordagem para execução dos testes no nível PIM, assim como uma evidência de teste. Para esta avaliação foi usada uma ferramenta open source chamada fUML Reference Implementation (MODELDRIVEN, 2010).

6. Analisar a escalonabilidade durante a avaliação de aplicabilidade;

O capítulo anterior apresentou a abordagem para análise de escalonabilidade, a partir de um modelo AADL gerado automaticamente usando-se a ferramenta Cheddar (LISYC, 2010).

7. Gerar um código executável, em Java, durante a avaliação de aplicabilidade;

O capítulo anterior apresentou a abordagem para geração automática de código a partir de um modelo PSM completo.

5.2. Fragilidades

Um resultado importante do presente trabalho foi localizar fragilidades na abordagem proposta pela MDA, tais fragilidades estão aqui documentadas para fomentar a avaliação da comunidade.

As fragilidades localizadas são causadas pelo risco de um foco demasiado na especificação de detalhes de baixo nível no PIM, como por exemplo, usar atividades para descrever equações.

Devido à quantidade de detalhes necessários na modelagem do PIM, é possível que o especialista envolvido perca a noção do todo e se preocupe em realizar uma boa especificação.

Outro problema é a especificação de detalhes desnecessários no PIM, ou melhor, que poderiam ser melhores descritos no PDM. Com o advento da fUML a linha que divide análise, projeto e implementação fica cada vez menos rígida, assim é possível que em algumas situações a fUML seja aplicada na modelagem de elementos que não precisariam ser definidos no PIM.

O último problema é que o especialista responsável por desenvolver uma plataforma (PDM + Transformações) pode ficar tentado a considerar apenas elementos disponíveis na plataforma alvo para fins de transformação, limitando a abordagem a ser usada para a definição do PIM.

5.3. Limitações

A seguir são listadas as limitações da abordagem, das ferramentas e dos protótipos criados no presente trabalho:

- Limitações da abordagem – são aquelas decorrentes da MDA e suas especificações.
    - Complexidade dos diagramas de atividades para a definição de comportamento no PIM – como fUML só permite a definição de comportamento através de atividades e não define uma sintaxe concreta textual para tais atividades, a única sintaxe concreta disponível é a definida pela UML, ou seja, diagrama de atividades. Infelizmente, para atividades maiores, tais diagramas rapidamente se tornam grandes, ilegíveis e difíceis de ser compreendidos (OMG, 2009c);
    - Impossibilidade de modelagem de tipos reais com fUML – a fUML só define os seguintes tipos básicos: *String, Integer, Boolean e UnlimitedNatural*. Uma necessidade recorrente, tanto em sistemas embarcados quanto em sistemas não embarcados, é um tipo que permita casas decimais. A fUML deixa a critério de cada ferramenta que implemente um interpretador a possibilidade de estendê-la com mais tipos básicos;
    - Dificuldade para modelar sistemas não baseados em eventos, ou contínuos. Um sistema mecânico como o controle de atitude da PMM assume uma complexidade muito maior se modelado usando-se fUML do que se modelado com ferramentas da engenharia de controle, ou expressões MathML usadas nos diagramas paramétricos da SysML;
- Limitações das ferramentas – são aquelas decorrentes das ferramentas empregadas na avaliação de aplicabilidade.
    - Grande dificuldade para desenvolvimento e depuração de modelos PIM – como as ferramentas estão em um estágio inicial

de desenvolvimento elas não permitem depuração e a exibição de erros exige um grande esforço para sua interpretação;

- Grande dificuldade para desenvolvimento e depuração de transformações MOF QVT e MOF M2T– como as ferramentas estão em um estágio inicial de desenvolvimento elas não permitem depuração e a exibição de erros exige um grande esforço para sua interpretação;

  MOF M2T não suporta o bloco referente à rastreabilidade, *trace*, o que é mais um ponto para dificultar a localização de eventuais erros;
- O quadro de trabalho EMF tem grandes dificuldades para manipular atributos multivalorados de estereótipos a partir de uma transformação MOF QVT, os quais são intensamente utilizados pelo UML *profile* MARTE – isso exige maior esforço no desenvolvimento de transformações de modelo para modelo usando a especificação MOF QVT;

- Limitações dos protótipos criados – são aquelas decorrentes dos protótipos desenvolvidos no presente trabalho.
  - Transformação inicial para AADL – a transformação para AADL gera apenas os elementos necessários para uma avaliação de escalonabilidade, ela não tem a pretensão de descrever todo o PSM usando AADL;
  - Transformação para AADL e protocolo de escalonamento – só permite o uso do protocolo *rate monotonic*;
  - Transformação para AADL – assume que todas as tarefas são periódicas.

6 CONCLUSÕES

A definição de uma abordagem assim como o desenvolvimento de protótipos para uma plataforma, que permita verificação em um nível independente de plataforma e análise de escalonabilidade para uma plataforma, pode ser considerado um primeiro passo no sentido de aumentar o uso de engenharia conduzida por modelos nos futuros satélites do INPE.

A seguir são apresentadas as principais contribuições deste trabalho, os possíveis trabalhos futuros assim como as considerações finais.

6.1. Contribuições para a comunidade em geral

Um ponto importante do presente trabalho é seu forte uso de especificações do OMG e de ferramentas que permitem contribuição (um dos critérios para a seleção das ferramentas para o presente trabalho), assim caso fossem localizados problemas ou oportunidades de melhorias estes poderiam ser submetidos e usados como *realimentação* pelas entidades responsáveis para futuras versões das especificações ou ferramentas.

Nesta linha, foram localizados durante o presente trabalho pontos de melhoria na especificação fUML que foram submetidos para análise pelo OMG, sendo eles:

1. Foi sugerida a alteração das seções 8.2.2.2 e 8.2.2.3 do documento de especificação da fUML para que o objeto corrente, contexto, seja enviado para o responsável por avaliar ações do tipo *ValueSpecificationAction*. O objetivo é permitir que sejam definidas linguagens, como a VSL do MARTE, que permitem a definição de equações de forma mais simples. Abaixo segue em negrito as sugestões.

8.2.2.2 ExecutionFactory

[3] createEvaluation (in specification : ValueSpecification, ***in context: Object[0..1]*** *) : Evaluation*

8.2.2.3 Executor

[1] evaluate (in specification : ValueSpecification, ***in context: Object[0..1]****) : Value*

return this.locus.factory.createEvaluation(specification, ***context****).evaluate();*

2. Foi sugerida a adição na seção 9.1 *Primitive Types,* do documento de especificação da fUML, do tipo Real, seguindo as mesmas características de tal tipo básico no UML *profile* MARTE.

A próxima seção discorre sobre as contribuições acadêmicas.

6.2. Contribuições acadêmicas

Esta seção apresenta as contribuições obtidas com o presente trabalho. As contribuições são apresentadas destacando-se os trabalhos correlatos e as principais diferenças.

- O corrente trabalho tem as seguintes contribuições quando comparado à Silva (2005):
  - É focado em sistemas embarcados de tempo real;
  - Existência do modelo PSM, ou seja, uso do mapeamento explícito (vide vantagens nas seções “2.8.5 MOF M2T” e “3.1 Considerações para concepção da proposta”);
  - Define comportamento nos modelos PIM e nos modelos PSM usando especificações do OMG (vide vantagens na seção “2.8 Especificações OMG”);
  - Protótipo das transformações implementado seguindo especificações do OMG (MOF QVT e MOF M2T) (vide vantagens na seção “2.8 Especificações OMG”);

- A transformação do PSM para código gera comportamento;

- O corrente trabalho tem as seguintes contribuições quando comparado à Moreira (2006):
  - Uso de técnicas definidas para tratar o problema de software, possibilitando, em futuros trabalhos, a integração com ambientes da engenharia de controle (vide vantagens na seção “2.5 Abordagens para aplicação de MDE”);
  - Capacidade maior para testar funcionalmente os modelos PIM (no trabalho de Moreira (2006) os testes funcionais independentes de plataforma eram realizados somente nos algoritmos de controle);
  - Capacidade para avaliar escalonabilidade do modelo PSM levando em consideração as características da plataforma de execução;
  - Ser totalmente independente de uma ferramenta ou fornecedor, já que se baseia em especificações do OMG, e mais, usa ferramentas *Opensource*;

- O corrente trabalho tem as seguintes contribuições quando comparado à Wehrmeister (2009):
  - Usa abordagem orientada a eventos em detrimento do paradigma “evento-Ação”, destaca-se isso pelo uso de diagramas de atividades descrevendo máquinas de estado em detrimento de diagramas de seqüência (vide vantagens na seção “2.7 Abordagens para modelagem de sistemas orientados a eventos”);
  - Capacidade para modelar subsistemas no nível PIM em paralelo (vide vantagens na seção “3.3.1 PIM”);
  - Uso de especificação do OMG (fUML) para definir comportamento nos modelos PIM em detrimento de construções específicas (vide vantagens na seção “2.8 Especificações OMG”);
  - Capacidade para testar funcionalmente os modelos PIM;

- Uso no PIM apenas do pacote HLAM do UML *profile* MARTE; Não são definidas no PIM as tarefas (*SchedulableResource*) apenas as propriedades não funcionais (vide vantagens na seção “2.8.7 fUML”);
- Existência de um modelo PDM descrevendo hardware e software com o UML *profile* MARTE (vide vantagens na seção “3.1 Considerações para concepção da proposta”);
- Existência do modelo PSM, ou seja, uso do mapeamento explícito (vide vantagens nas seções “2.8.5 MOF M2T” e “3.1 Considerações para concepção da proposta”);
- Capacidade para avaliar escalonabilidade do modelo PSM levando em consideração as características da plataforma de execução;
- Protótipo das transformações implementado seguindo especificações do OMG (MOF QVT e MOF M2T) (vide vantagens na seção “2.8 Especificações OMG”);

- O corrente trabalho tem as seguintes contribuições quando comparado à Feiler et al. (2007):
  - Geração automática do modelo PSM usando-se UML e MARTE, em detrimento a AADL;
  - Uso de especificação do OMG (fUML) para definir comportamento nos modelos PIM em detrimento de construções específicas (vide vantagens na seção “2.8 Especificações OMG”);
  - Protótipo das transformações implementado seguindo especificações do OMG (MOF QVT e MOF M2T) (vide vantagens na seção “2.8 Especificações OMG”);

- O corrente trabalho tem as seguintes contribuições quando comparado à Maes (2007):

- Transformação de um modelo UML anotado com o MARTE para AADL e não para um formato proprietário de uma ferramenta (vide vantagens na seção “2.8 Especificações OMG”);
- Não utiliza o pacote SAM do MARTE, mas sim os pacotes para modelagem de hardware e software (vide vantagens na seção “3.3.5 AADL”);

- O corrente trabalho ainda tem as seguintes contribuições não localizadas em outros trabalhos:
  - Permitir uma visão detalhada de uma possível abordagem para uso da MDA, passando pelas especificações, o que usar de cada uma, por ferramentas que permitem o uso de tais especificações e finalizando com a apresentação de uma avaliação de aplicabilidade;
  - Uso da fUML para modelar e testar comportamento dos modelos PIM (vide vantagens na seção “2.8 Especificações OMG”);
  - Apresentar o processo de entrelaçamento de modelos integrado ao desenvolvimento de modelos, e de uma forma automática;
  - O presente trabalho permitiu questionar a visão apresentada por Mellor e Balcer (2002), onde no nível PIM são definidas pontes para subsistemas técnicos (no nível do PSM). Ou seja, o presente trabalho sugere que não exista nenhuma dependência entre o modelo PIM de um sistema e subsistemas no nível PSM, nem mesmo de pontes usando-se terminadores. A tecnologia de implementação deve ser definida em um nível inferior, usando-se transformações;
  - Existência do modelo PDM como um modelo independente (vide vantagens na seção “3.1 Considerações para concepção da proposta”);

        - Particionamento do modelo PDM em dois outros modelos, PDM Software e PDM Software – comunicação entre modelos;
        - Existência de uma parte do modelo PDM voltada para a modelagem do hardware embarcado, permitindo análise de hardware e software;
    - Combinar em uma única forma de modelar a abordagem síncrona e assíncrona para definição de sistemas de tempo real. A abordagem síncrona é usada no PIM através da fUML. A abordagem assíncrona é usada no PSM através do MARTE e da plataforma Java.

6.3. Futuros trabalhos

A conclusão deste trabalho não encerra, de forma alguma, as pesquisas a serem realizadas sobre o tema “engenharia conduzida por modelos aplicada a software de tempo real espacial”. Na verdade, este trabalho contribui para que uma série de outros trabalhos possam ser realizados.

A seguir segue uma breve lista de possíveis futuros trabalhos:

- A partir de um modelo PIM definido usando-se fUML é possível criar automaticamente cenários de teste. Existem pesquisas avaliando tal possibilidade, notadamente (LINZHANG et al., 2004) (CHEN e MISHRA, 2004);
- A determinação do *Worst Case Time Execution* é sempre um tema a ser explorado em sistemas de tempo real, é possível antever pesquisas focadas em definir este tempo baseando-se em simulação de hardware e software (OMG, 2009a) e não em uma base histórica, como sugerido no presente trabalho;
- Localizar *deadlocks* automaticamente através de análise do PSM;

- Avaliar critérios para analisar uma solução PSM candidata, e permitir a otimização do PSM;
- Aprimorar a abordagem sugerida para entrelaçamento de modelos, para que um meta-modelo de entrelaçamento seja uma entrada adicional a transformação sugerida, permitindo maior flexibilidade e uso de classes abstratas;
- Usar melhor o potencial do MARTE. Um exemplo é gerar automaticamente diagramas de seqüência no PSM, anotados com o pacote SAM do MARTE, permitindo análise de escalonabilidade de forma mais profunda e rigorosa. Outro exemplo é o uso do pacote de alocação, similar ao conceito de alocação da SysML;
- Explorar como integrar a engenharia de sistemas, mais especificamente sua linguagem padrão (SysML, *System Modeling Language*), com o UML *profile* MARTE (Espinoza, 2008) e a abordagem MDA. Um ponto claro é que seria possível gerar automaticamente diagramas paramétricos a partir de diagramas de atividades de restrições, ou vice-versa.
- Uso de uma plataforma alvo que possua as características necessárias para um sistema de tempo real, destaca-se a plataforma Java Real Time;
- Avaliar a abordagem proposta neste trabalho de forma mais profunda, avaliando-se um sistema mais completo e com uma linguagem que respeite as necessidades de um sistema de tempo real;
- Explorar MDA na engenharia de sistemas (CLOUTIER, 2006) (INCOSE, 2008) e o cruzamento de modelagem de software com a engenharia de sistemas (HAUSSE e THOM, 2008);
- Explorar o modelo mais abstrato sugerido pela MDA, o modelo independente de computadores (CIM, *Computational Independent Model*), como o descrever de forma que seja possível geração de parte do PIM ou uso das informações do CIM no PIM;

- Explorar como o tópico tolerância a falhas pode ser explorado usando-se MDA. Foi citado neste trabalho que a técnica “Redundância através de Projeto Diverso” pode sofrer grandes modificações com o advento da MDA, outras técnicas como redundância homogênea (*Homogeneous Redundancy*) podem sofrer alterações, como o exemplo encontrado em Mellor e Balcer (2002) que cita uma transformação capaz de gerar um PSM tolerante a falhas através desta técnica;
- Avaliar estratégias e padrões para transformações de PIM para PSM. Um exemplo é a simplificação realizada no presente trabalho, toda classe ativa se transforma em uma tarefa, isto pode não se aplicar em casos onde o número de classes ativas é muito alto ou onde o tempo para se chavear entre tarefas seja muito significativo. Outro exemplo é a tradução das atividades para código; Outro é como integrar quadros de trabalho ou subsistemas no nível PSM de forma declarativa, por exemplo, como integrar um sistema gerenciador de banco de dados em uma transformação de forma declarativa; Outro ainda é como parametrizar transformações, por exemplo: como trocar o modelo que descreve a comunicação entre objetos sem ter que alterar a transformação ou a biblioteca *blackbox*;
- Por último, é possível vislumbrar uma ferramenta que interprete fUML apresentando os resultados de forma gráfica, permitindo a depuração de um modelo PIM definido com fUML. Tal ferramenta deveria ser construída como um *plugin* do Eclipse (ECLIPSE FOUNDATION, 2010) e deveria permitir a integração com o Matlab. Ela poderia ainda avaliar consistência dos modelos PIM, como por exemplo: toda a classe ativa deve ter um *ClassifierBehavior*, todas as ações *SendSignalAction* possuem pelo menos um *AcceptEventAction* relacionado, não devem existir loops infinitos, existência de condições que nunca são satisfeitas, etc...

6.4. Considerações finais

Existe muito a ser explorado sobre o tema selecionado pelo presente trabalho, a seção anterior mostrou alguns possíveis tópicos, mas uma breve consulta a órgãos como o SEI (*Software Engine Institute*) mostra o quanto o tema tem sido explorado. Feiler e Niz (2008) destacam um estudo de 2007 da Academia Nacional de Ciências dos Estados Unidos (*National Academy of Sciences*) sobre certificação de sistemas, que determina que é o momento para se investir em abordagens mais formais para a engenharia de sistemas embarcados aplicando-se análise quantitativa sobre modelos da arquitetura do sistema.

Um aspecto não profundamente explorado no presente trabalho é quanto ao real benefício do uso de uma abordagem similar a proposta, pois ela está fortemente pautada em reuso. Entretanto o reuso está associado à criação de elementos flexíveis, que por sua vez exigem um investimento inicial superior. As principais perguntas são: Vale a pena usar uma abordagem similar a proposta? Em que casos ela é aplicável?

O presente trabalho não tem a pretensão de definir um método, muito menos um processo para o uso da MDA. Ele refina a abordagem proposta pela MDA transformando em uma abordagem factível (porque usar, o que usar, como usar e quando usar) essencialmente baseada na engenharia de sistemas, já que de uma forma ou de outra o presente trabalho atua em pelo menos três áreas da engenharia: software (primariamente), controle e hardware.

REFERÊNCIAS BIBLIOGRÁFICAS

AGÊNCIA ESPACIAL BRASILEIRA (AEB). **Programa Nacional de Atividade Espaciais 2005-2014**: PNAE. 2005. Brasília: Ministério da Ciência e Tecnologia.

BRIAND, L. P.; ROY, D. M. **Meeting deadlines in hard real-time systems – the rate monotonic approach**. USA: Institute of Eletrical and Electronics Engineers, Inc, 1999. 253 p. ISBN 0-8186-7406-7.

CEA. **Papyrus Site**. 2010. Disponível em: <http://www.papyrusuml.org/>. Acesso em: 03 jun. 2010.

CLOUTIER, R. **MDA for systems engineering – why should we care?** USA: 2006. 10 p. Disponível em: <http://www.calimar.com/Papers/Model%20Driven%20Architecture%20for%20SE-Why%20Care.pdf>. Acesso em: 25 jun. 2010.

DORF, R. C.; BISHOP, R. H. **Modern control systems**. 10. ed. New York: Prentice Hall, 2005. 717 p.

DOUGLAS, B. P. **Doing hard time**: developing real-time systems with UML, objects, frameworks, and patterns. New York: Addison Wesley, 1999. 749 p. ISBN 0-201-49837-5.

DVORAK, R. **Model Transformation with Operational QVT** - QVT Operational - M2M component. USA, 2008. EclipseCon 2008. Disponível em: <http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf>. Acesso em: 25 jun. 2010.

ECLIPSE FOUNDATION. **ATL transformation site**. 2010a. Disponível em: <http://www.eclipse.org/m2m/atl/atlTransformations/>. Acesso em: 25 jun. 2010.

ECLIPSE FOUNDATION. **Eclipse site**. 2010b. Disponível em: <http://www.eclipse.org>. Acesso em: 25 jun. 2010.

ECLIPSE FOUNDATION. **Eclipse modeling framework technology site**. 2010c. Disponível em: < http://www.eclipse.org/modeling/emft/>. Acesso em: 25 jun. 2010.

ECLIPSE FOUNDATION. **Eclipse model driven tools site**. 2010d. Disponível em: <http://www.eclipse.org/modeling/mdt >. Acesso em: 25 jun. 2010.

FARINES, J.; FRAGA, J. S.; OLIVEIRA, R. S. **Sistemas de Tempo Real**. São Paulo: IME-USP, 2000. 201 p.

FEILER, P.; GLUCH, D.; HUDAK, J. **The Architecture Analysis & Design Language (AADL):** an introduction. USA: Software Engineering Institute, Carnegie Mellow, 2006. 144 p. Disponível em: <http://www.sei.cmu.edu/library/abstracts/reports/06tn011.cfm>. Acesso em: 25 jun. 2010. CMU/SEI-2006-TN-011

FEILER, P.; NIZ, D.; RAISTRICK, C.; LEWIS, B. From PIMs to PSMs. In: IEEE INTERNATIONAL CONFERENCE ON ENGINEERING COMPLEX COMPUTER SYSTEMS, 23, 2007, Auckland, New Zealand. **Proceedings…** USA: IEEE, 2007. p. 365–370. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/ICECCS.2007.25>. Acesso em: 26 mar. 2010.

FEILER, P.; NIZ, D. **ASSIP study of real-time safety-critical embedded software-intensive system engineering practices.** USA: Software Engineering Institute, Carnegie Mellow, 2008. 50 p. Disponível em: <http://www.sei.cmu.edu/library/abstracts/reports/08sr001.cfm>. Acesso em: 25 jun. 2010. CMU/SEI-2008-SR-001

FLINT, S. R.; BOUGHTON, C. B. **Executable/translatable UML and systems engineering.** Department of Computer Science, Australian National University, 2003. 14 p. Disponível em: <http://www.seecforum.unisa.edu.au/sete2003/papers%20&%20presos/Flint_Shayne%20and%20Boughton_Clive_PAPER.pdf>. Acesso em: 25 jun. 2010.

FOWLER, M. UML **Essencial um breve guia para a linguagem-padrão de modelagem de objetos**. 3. Ed.. Brasil: Bookman Companhia Editora, 2005. 160 p. ISBN 85-363-0454-5.

GOMAA, H. **Designing concurrent, distributed and real-time applications with UML**. USA: Addison Wesley, 2000. 785 p. ISBN 0-201-65793-7.

HOLT, J.; PERRY, S. **SysML for systems engineering**. London: The Institution of Engineering and Technology, 2008. 335 p. ISBN 978-0-86341-825-9.

INTERNATIONAL BUSINESS MACHINES (IBM) **rational software architect site.** 2010. Disponível em: <http://www.ibm.com>. Acesso em: 02 jun. 2010.

INTERNATIONAL COUNCIL ON SYSTEMS ENGINEERING (INCOSE). **Survey of Model-Based Systems Engineering (MBSE) methodologies**. USA: INCOSE, Seattle, 2008. 80 p. Disponível em: <http://www.incose.org/productspubs/pdf/techdata/MTTC/MBSE_Methodology_Survey_2008-0610_RevB-JAE2.pdf>. Acesso em: 25 jun. 2010.

KENNEDY CARTER. **Supporting model driven architecture with eXecutable UML**. Kennedy Carter. USA: Kennedy Carter, 2002. 44 p. Disponível em: <http://www.kc.com>. Acesso em: 02 fev. 2010. Ref: CTN 80 v 2.2.

KENNEDY CARTER. **UML ASL reference guide**: ASL language level 2.5. USA: Kennedy Carter, 2003. 112 p. Disponível em: <http://www.kc.com>. Acesso em: 02 fev. 2010. Manual Revision D.

KOPETZ, H. **Real-time systems** – Design Principles for Distributed Embedded Applications. USA: Kluwer Academic Publishers, 1997. 338 p.

LEWIS, B.; FEILER, P. **Impact of architectural model-based engineering with AADL**. USA: Software Engineering Institute, Carnegie Mellow, 2007. 38 p. Disponível em: <http://csse.usc.edu/csse/event/2009/ARR/presentationByDay/Wed_Mar18/3_MBE-AADL-USC%20March%202009.ppt>. Acesso em: 25 jun. 2010.

LISYC. **Cheddar site**. LISyC, Université de Brest. 2010. Disponível em: <http://beru.univ-brest.fr/~singhoff/cheddar/index-fr.html>. Acesso em: 25 jun. 2010.

MAES, E. **Validation de systèmes temps-réel et embarqué à partir d'un modèle MARTE**. France: Thales Research and Technology, 2007. 24 p. Disponível em: <http://neptune.irit.fr/images/files/Neptune2008/Transparents/P03_PR_EMaes.pdf>. Acesso em: 02 fev. 2010.

MAES, E. VIENNE, N. **MARTE to Cheddar transformation using ATL**. France: Thales Research and Technology, 2007. 80 p. Disponível em: <http://beru.univ-brest.fr/svn/CHEDDAR-2.0/trunk/contribs/examples_of_use/thales_rt/MARTE2CheddarTransformationRules.pdf >. Acesso em: 02 fev. 2010. Document Number: 61565546-179.

MARCOS, D. D. F.; JEAN, B.; FRÉDÉRIC, J.; ERWAN, B.; GUILLAUME, G. AMW: A Generic Model Weaver. In: L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES (IDM05), 2005, Paris, France. **Electronic Proceedings…**Paris: Premières Journées, 2005. 10 p. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.1294&rep=rep1&type=pdf>. Acesso em: 02 jun. 2010.

MATHWORKS. **Matlab site**. 2010a. Disponível em: <http://www.mathworks.com/products/matlab/>. Acesso em: 20 ago. 2010.

MATHWORKS. **MATRIXx tech note**. 2010b. Disponível em: <http://www.mathworks.com/support/tech-notes/2200/2201.html>. Acesso em: 20 ago. 2010.

MELLOR, S. J.; BALCER, M. J. **Executable UML**: a foundation for model-driven architecture. USA: Addison Wesley, 2002. 416 p. ISBN 0-201-74804-5.

MODELDRIVEN. **FUML reference implementation site**. 2010. Disponível em: <http://www.modeldriven.org>. Acesso em: 01 jun. 2010.

MOREIRA M. L. B. **Projeto e simulação de um controle discreto para a plataforma multi-missão e sua migração para um sistema operacional de tempo real**. 2006. 181 p. (INPE-14202-TDI/1103). Dissertação (Mestrado em Mecânica Espacial e Controle) - Instituto Nacional de Pesquisas Espaciais, São José dos Campos, Brasil. 2006. Disponível em: <http://urlib.net/rep/sid.inpe.br/MTC-m13@80/2006/07.10.13.42?languagebutton=pt-BR>. Acesso em: 10 out. 2009.

MRAIDHA, C. **Model transformations and code generation**. 2006. France: Ecole IN2P3 Temps Réel, CEA List, 2009. 17 p. Disponível em: <http://www.in2p3.fr/actions/formation/Info09/TPC_Chokri.pdf>. Acesso em: 02 mar. 2010.

NOMAGIC. **MagicDraw site**. 2010. Disponível em: <http://www.magicdraw.com >. Acesso em: 02 jun. 2010.

NASS, R. **An insider's view of the 2008 embedded market study**. **Embedded Systems Design,** San Francisco, v.21, n.9, Sept. 2008. Disponível em: <http://www.embedded.com/design/testissue/210200580>. Acesso em: 09 set. 2009.

OBEO. **Acceleo project site**. 2010. Disponível em: <http://acceleo.org/pages/home/>. Acesso em: 25 jun. 2010.

OGATA, K. **Modern control engineering**. 3. ed. USA: Prentice Hall, 1997. 987 p. ISBN 0-13-227307-1.

OBJECT MANAGEMENT GROUP (OMG). **Model-driven architecture**. Needham, MA , USA, 2003. 62 p. Disponível em: <http://www.omg.org/mda>. Acesso em: 17 mai. 2009.

OBJECT MANAGEMENT GROUP (OMG). **Meta Object Facility (MOF) core specification**: Version 2.0. Needham, MA , USA, 2006. 88 p. Disponível em: <http://www.omg.org/mof/>. Acesso em: 18 fev. 2010.

OBJECT MANAGEMENT GROUP (OMG). **MOF 2.0/XMI mapping**: Version 2.1.1. Needham, MA , USA, 2007. 120 p. Disponível em: <http://www.omg.org/technology/documents/formal/xmi.htm>. Acesso em: 17 mai. 2009.

OBJECT MANAGEMENT GROUP (OMG). **Meta Object Facility (MOF 2.0)** Query/View/Transformation Specification: Version 1.0. Needham, MA , USA, 2008a. 240 p. Disponível em: <http://www.omg.org/spec/QVT/1.0/>. Acesso em: 18 fev. 2010.

OBJECT MANAGEMENT GROUP (OMG). **MOF Model to text transformation language**: Version 1.0. Needham, MA , USA, 2008b. 48 p. Disponível em: <http://www.omg.org/spec/MOFM2T/1.0/>. Acesso em: 15 set. 2009.

OBJECT MANAGEMENT GROUP (OMG). **UML Profile for MARTE**: mdeling and Analysis of Real-Time Embedded Systems: Version 1. Needham, MA , USA, 2009a. 738 p. Disponível em: <http://www.omgmarte.org/>. Acesso em: 02 mar. 2010.

OBJECT MANAGEMENT GROUP (OMG). **Unified modeling language, superstructure**: Version 2.2. Needham, MA , USA, 2009b. 740 p. Disponível em: <http://www.uml.org/>. Acesso em: 15 mar. 2009.

OBJECT MANAGEMENT GROUP (OMG). **Semantics of a foundational subset for executable UML models:** Version FTF Beta 2. Needham, MA , USA, 2009c. 349 p. Disponível em: <http://www.omg.org/spec/FUML/>. Acesso em: 09 fev. 2010.

OBJECT MANAGEMENT GROUP (OMG). **Unified modeling language, infrastructure**: Version 2.2. Needham, MA , USA, 2009d. 226 p. Disponível em: <http://www.uml.org/>. Acesso em: 15 mar. 2009.

PASETTI, A.; BROWN, T. **Software frameworks and embedded control systems**. UK: Springer, 2002. 293 p. ISBN: 978-3-540-43189-3.

PASETTI, A. **AOCS framework project site**. 2002. Disponível em: <http://control.ee.ethz.ch/~ceg/AocsFramework/FutureDevelopments.html#Java Implementation>. Acesso em: 02 fev. 2010.

PRESSMAN, R. S. **Engenharia de software**. 6. ed. São Paulo: Mc Graw Hill, 2006. 720 p. ISBN 85-86804-57-6.

SAMEK, M. **Practical UML statecharts in C/C++.** event-driven programming for embedded systems. 2. Ed..USA: Elsevier, Inc, 2009. 712 p. ISBN 978-0-7506-8706-5.

SILVA, M. B. R. **Uma avaliação da abordagem MDA através de um caso de uso.** 2005. 134 p. Dissertação (Mestrado) – Universidade Federal Fluminense, Rio de Janeiro, Brasil. 2005. Disponível em: <http://www.tempo.uff.br/arquivos/MBRS2005.pdf>. Acesso em: 21 dez. 2010.

STANKOKIC, J. A. Misconceptions about real-time computing. USA: Institute of Electrical and Electronics Engineers . **Computer**, v. 21, n. 10, p. 10-19, Oct 1988. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7053>. Acesso em: 30 nov. 2009.

SOUZA, M. O.; TRIVELATO, G. C. Simulators and simulations - their characteristics and applications to the simulation and control of aerospace vehicles. In: SAE BRAZIL CONGRESS, 12., São Paulo, SP, Brasil, 2004. **Proceedings…** Brasil: 2004. 10 p. Disponível em: <http://www2.dem.inpe.br/marcelo/dados/SAEBrasil2004paper279_part1.pdf >. Acesso em: 30 nov. 2009.

SOUZA, M. O.; CARVALHO, T. The Fault Avoidance and the Fault Tolerance Approaches for Increasing the Reliability of Aerospace and Automotive Systems. In: SAE BRAZIL CONGRESS, 12., São Paulo, SP, Brasil, 2005. **Proceedings...** Brasil: 2005. 16 p. Disponível em: <http://www2.dem.inpe.br/marcelo/dados/SAEBrasil2005paper328.pdf >. Acesso em: 30 nov. 2009.

TOPCASED. **TOPCASED site**. 2010. Disponível em: <http://http://www.topcased.org>. Acesso em: 05 jun. 2010.

VANDERPERREN, Y. DEHAENE, W. From UML/SysML to Matlab/Simulink: current state and future perspectives. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1, 2006, Munich, Germany. **Proceedings…** Germany: 2006. 1 p.

WANG, WENHENG. **Evaluation of UML model transformation tools**. 2005. 102 p. Thesis – Vienna University of Technology, Vienna, Austria. 2005. Disponível em: <http://www.big.tuwien.ac.at/teaching/theses/ma/wang.pdf>. Acesso em: 28 set. 2009.

WEHRMEISTER, M. A. **An aspect-oriented model-driven engineering approach for distributed embedded real-time systems**. 2009. 206 p. Tese (Doutorado) – Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil. 2009. Disponível em: <http://www.lume.ufrgs.br/handle/10183/17792>. Acesso em: 21 dez. 2009.

Bibliografia complementar

CHEN, M. MISHRA, P. KALITA, D. Coverage-driven automatic test generation for uml activity diagrams. In: ACM GREAT LAKES SYMPOSIUM ON VLSI, n. 18, 2008, USA. **Proceedings…** USA: ACM, 2008. p. 139 – 142. ISBN 978-1-59593-999-9.

ESPINOZA, H.; CANCILA, D. GÉRARD, S. **Challenges in combining SysML and MARTE for model-based design of embedded systems**. Paris: CEA, 2008. 18 p. Disponível em: <http://www.omg.org/cgi-bin/doc?syseng/09-06-08.pdf >. Acesso em: 04 mar. 2010. CEA internal report DRT/LIST/DTSI/SOL/08-276/HE

HAUSSE, M.; THOM, F**. building bridges between systems and software with SysML and UML.** USA: INCOSE, 2008. 18 p. Disponível em: < http://www.omgsysml.org/Building_SE-SW_Bridges%20.pdf>. Acesso em: 04 mar. 2010.

LINZHANG, W.; JIESONG, Y.; XIAOFENG, Y.; JUN, H.; XUANDONG, L.; GUOLIANG, Z. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, n. 11, 2004, Busan, Korea. **Proceedings…** Busan, Korea, IEEE Computer Society, 2004. p. 284 – 291. ISBN 0-7695-2245-9.

APÊNDICE A – AVALIAÇÃO DE APLICABILIDADE – SUBSISTEMA DE LEITURA DE SENSORES DA PMM

O presente apêndice tem como objetivo permitir ao leitor uma visão detalhada da avaliação de aplicabilidade realizada pelo presente trabalho. Ela tem um escopo extremamente reduzido versus a modelagem do sistema de controle de atitude e órbita da PMM realizado por Moreira (2006), dado que o foco da atual dissertação não é a modelagem do algoritmo(s) de controle da PMM, mas sim apresentar uma abordagem diferenciada pautada na MDA e especialmente em um método baseado em transformações.

Sobre o subsistema escolhido, leitura de sensores, já foram apresentados os critérios utilizados para a definição do que faria parte de tal subsistema.

Moreira (2006) modela os giroscópios com ganho unitário em relação à entrada e saída, mais um ruído aleatório. Um ganho unitário em relação à entrada e saída quer dizer que a voltagem lida no sensor é a velocidade angular do respectivo eixo. No presente trabalho, optou-se por retirar o ruído aleatório e manter-se o ganho unitário, entretanto optou-se por aplicar um deslocamento para que o valor zero não faça parte do intervalo válido de leitura.

Esta alteração foi feita com o objetivo de identificação de falhas, pois assim quando um giroscópio queimar será possível distinguir entre a inexistência de voltagem e um valor medido zero (KOPETZ, 1997).

A seguir são apresentados os modelos PIM para o subsistema de leitura de sensores e também para seu teste, assim como o modelo PSM gerado, o modelo AADL gerado e parte do código gerado, mais especificamente apenas para o gerenciador de sensores.

A.1 PIM – Subsistema de leitura de sensores da PMM

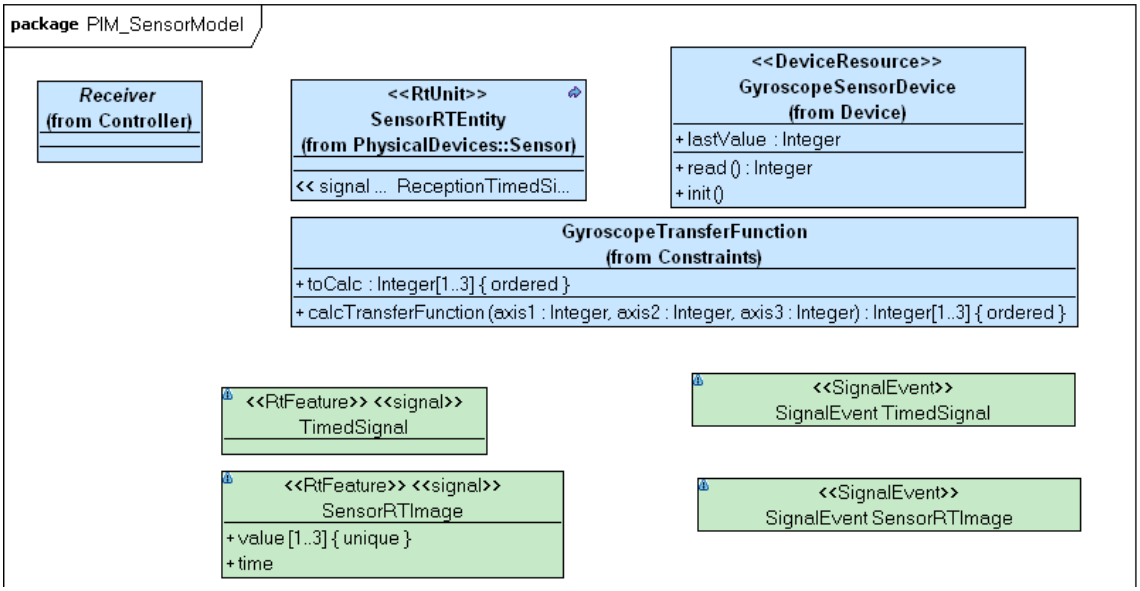

Figura A.1 - Modelagem da estrutura do subsistema de sensores da PMM usando diagrama de classes

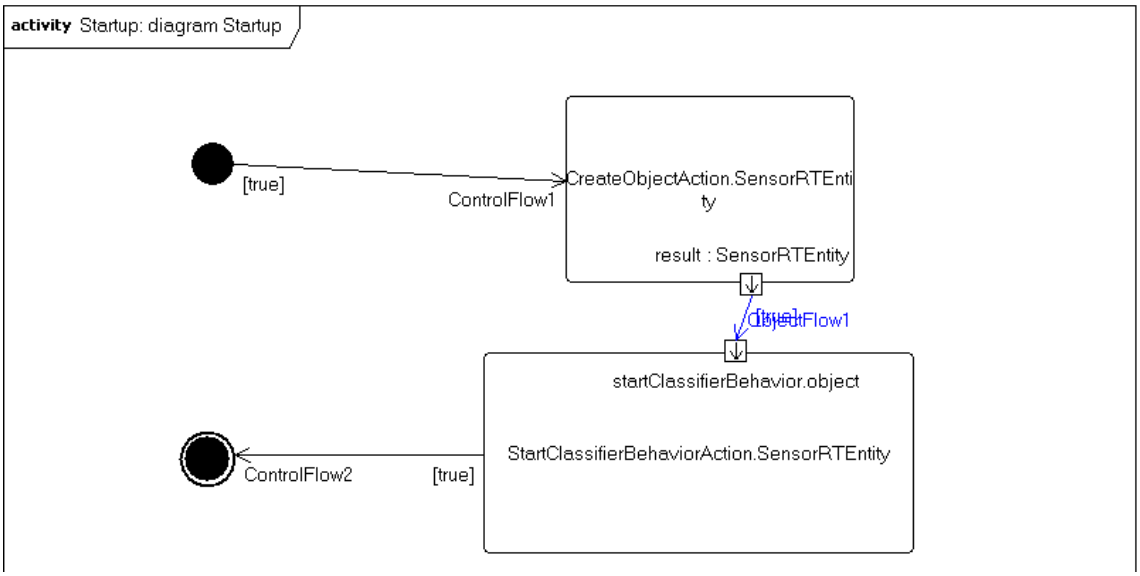

Figura A.2 - Modelagem da atividade de inicialização do subsistema de leitura de sensores

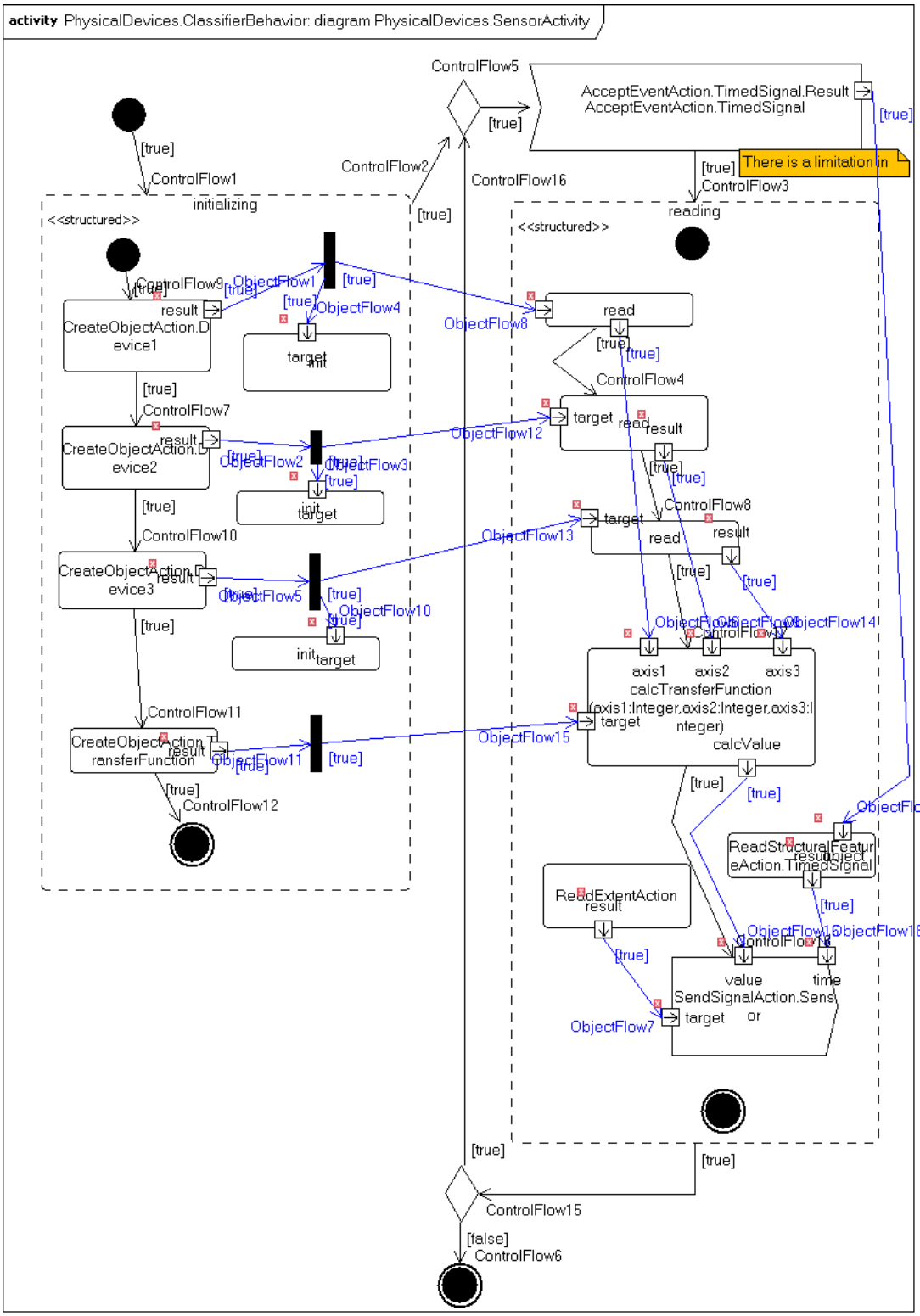

Figura A.3 - Modelagem de comportamento da classe *SensorRTEntity* usando diagrama de atividades para representar uma máquina de estados

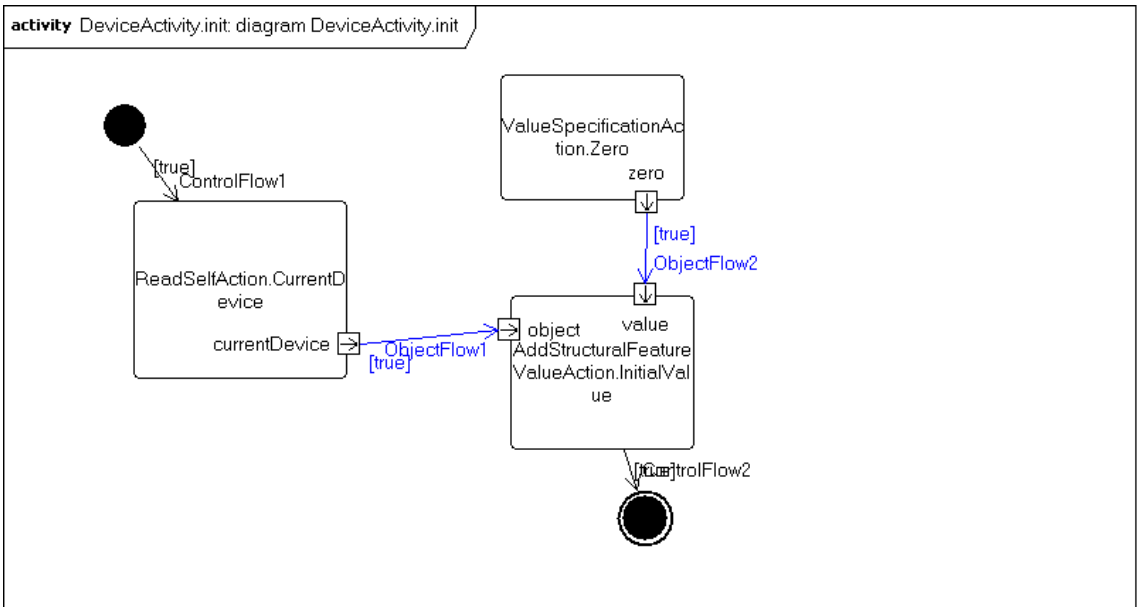

Figura A.4 - Modelagem de comportamento para a operação *init* do *GyroscopeSensorDevice*

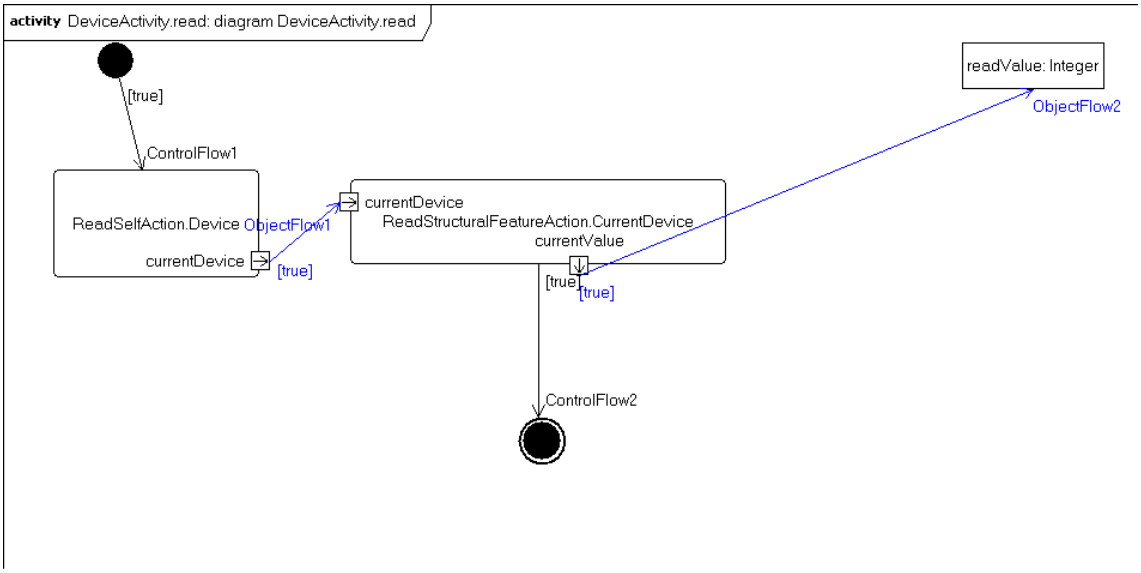

Figura A.5 - Modelagem de comportamento para a operação *read* do *GyroscopeSensorDevice*

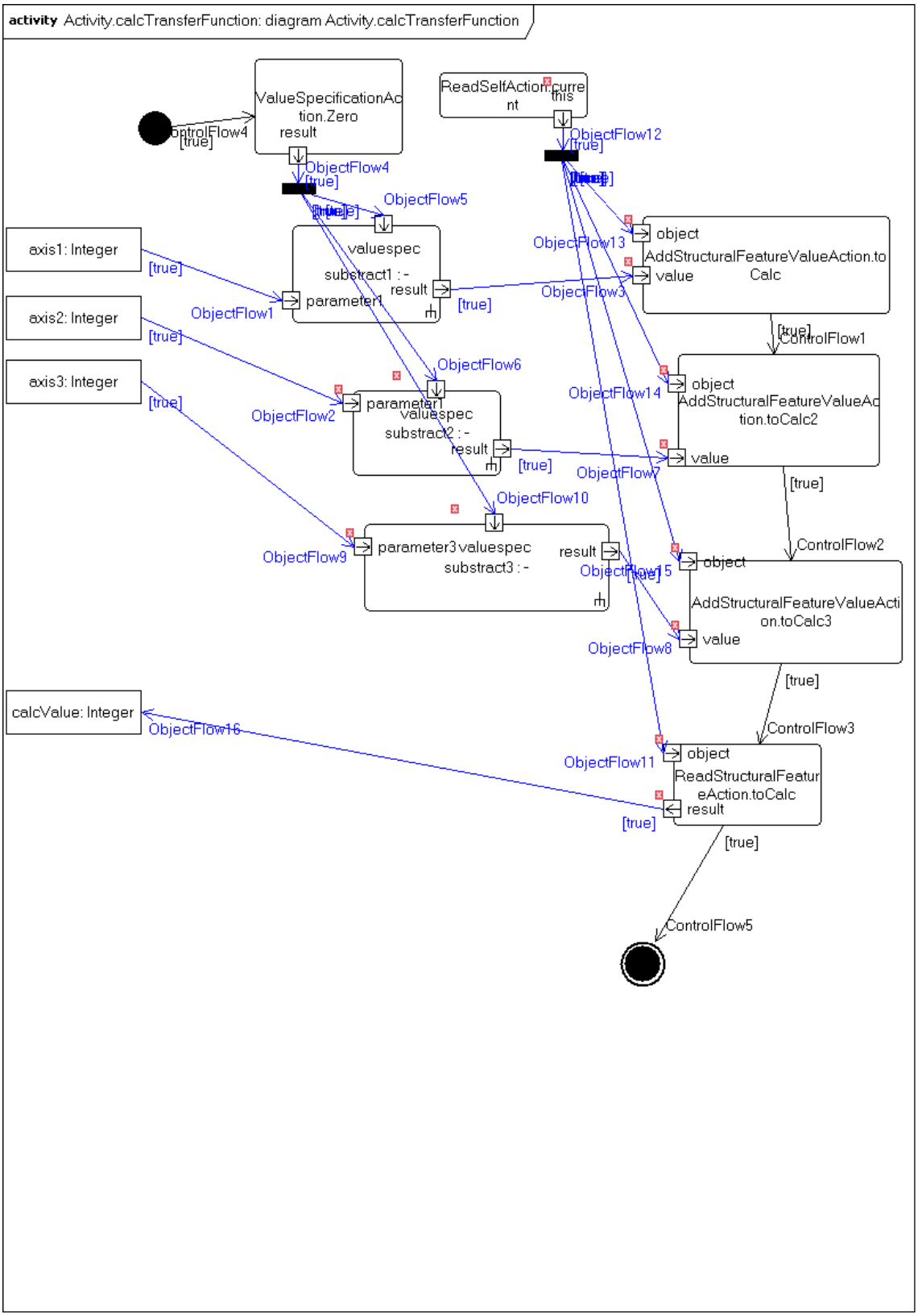

Figura A.6 - Modelagem de comportamento da operação *calcTransferFunction* do *GyroscopeTransferFunction*

A.2 PIM – Subsistema de leitura de sensores da PMM - Teste

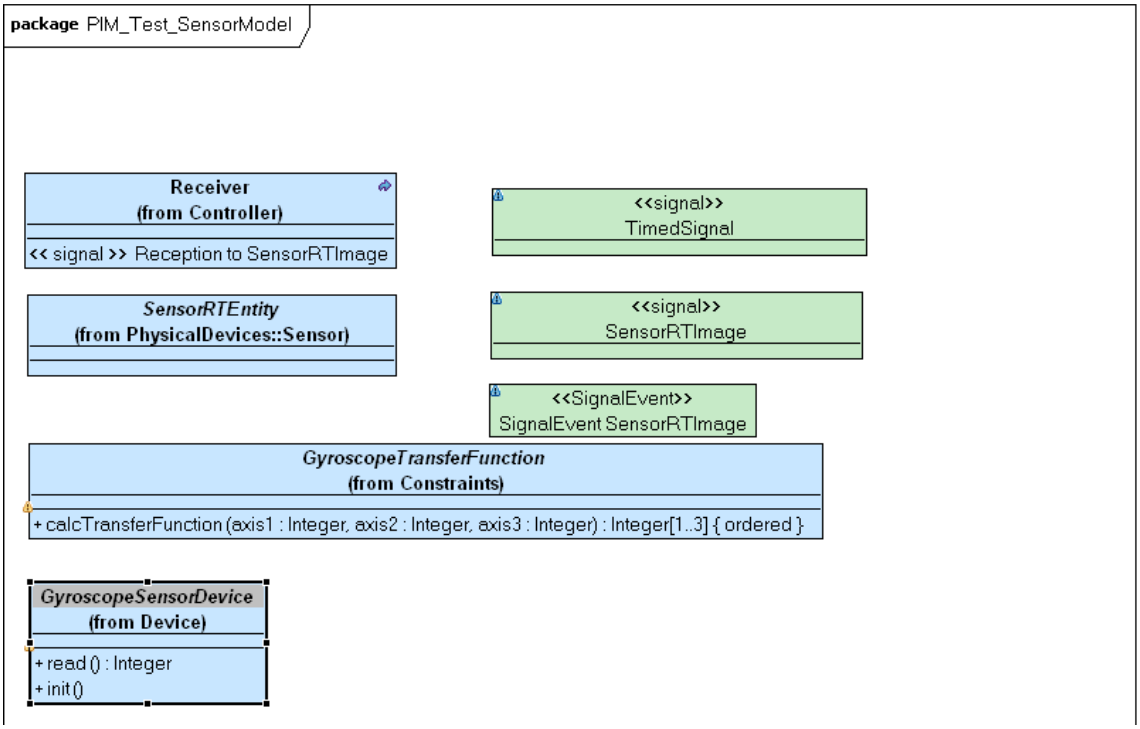

Figura A.7 - Modelagem da estrutura do modelo de teste do subsistema de sensores da PMM usando diagrama de classes

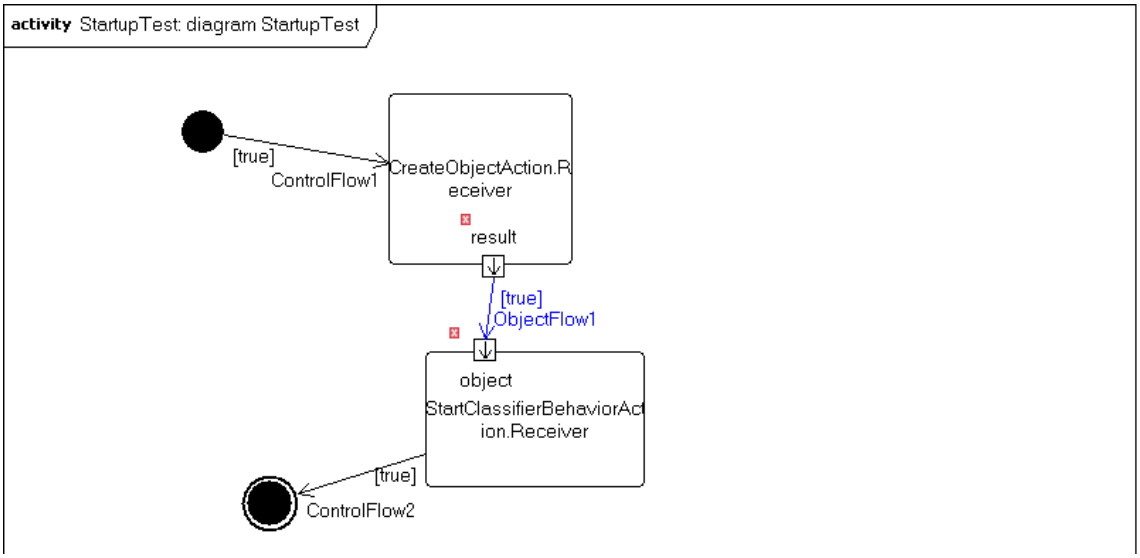

Figura A.8 - Modelagem da atividade de inicialização do modelo de teste do subsistema de leitura de sensores

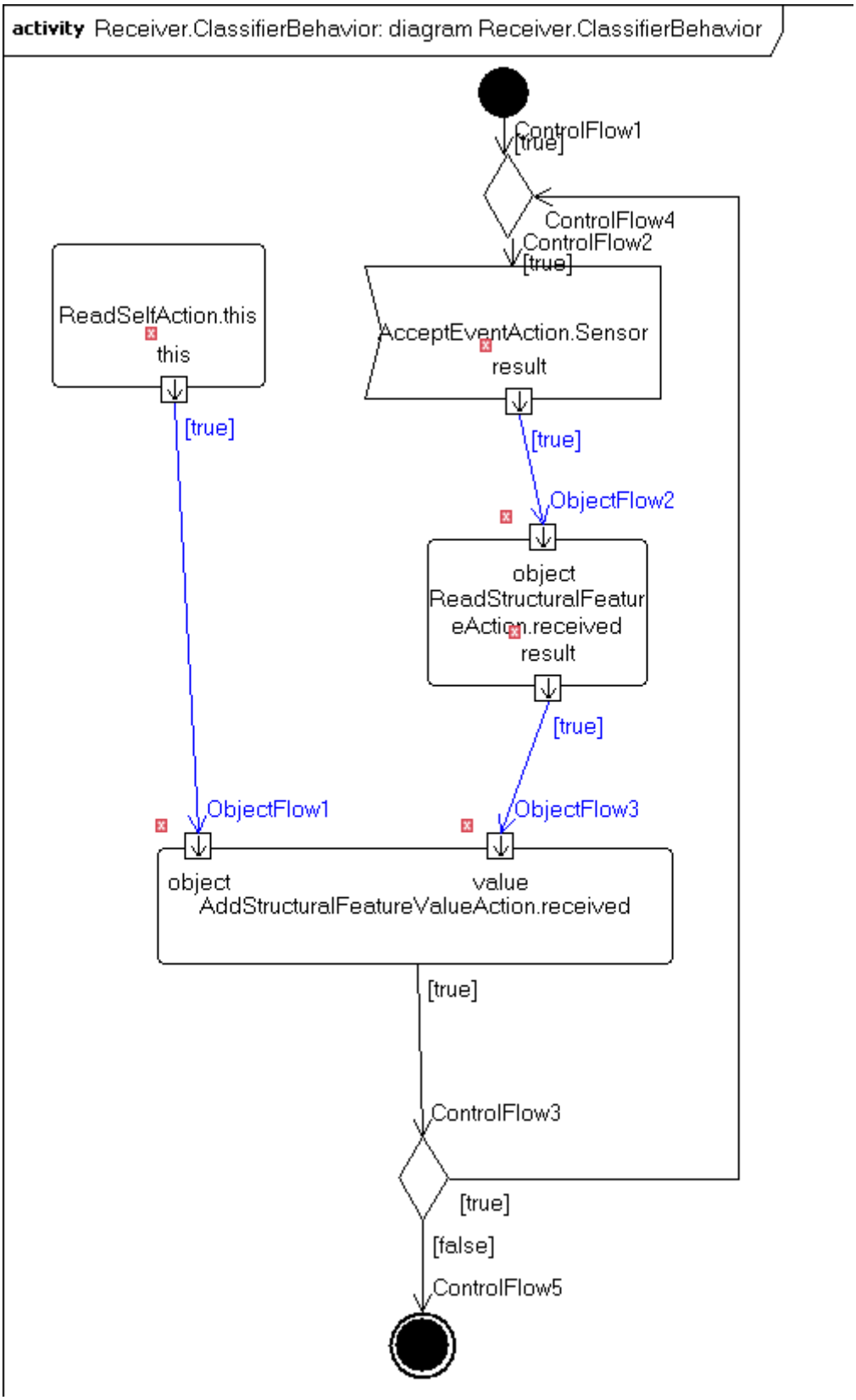

Figura A.9 - Modelagem de comportamento da classe *Receiver* usando diagrama de atividades

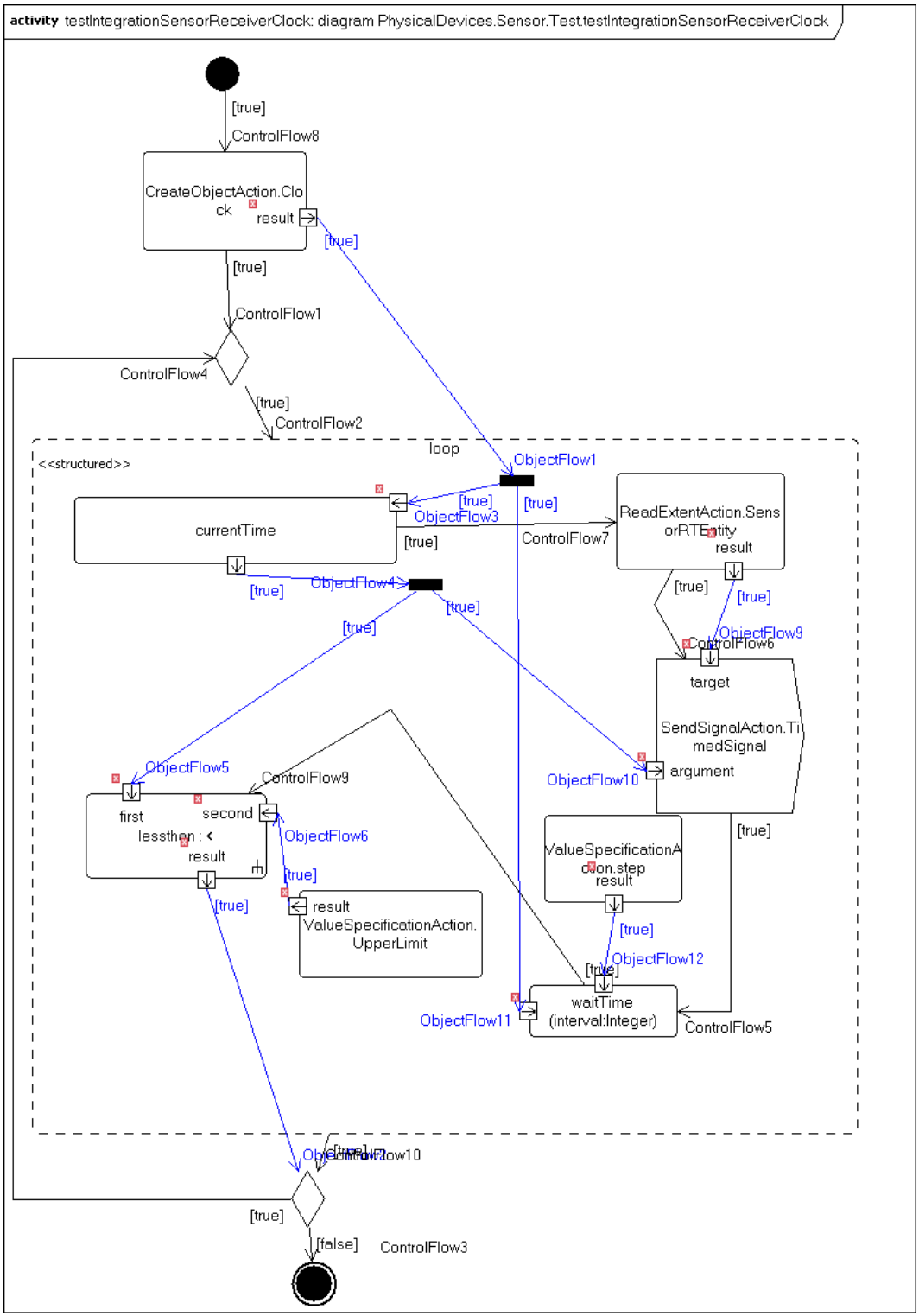

Figura A.10 - Modelagem de comportamento de um teste baseado no relógio para o subsistema de leitura de sensores da PMM

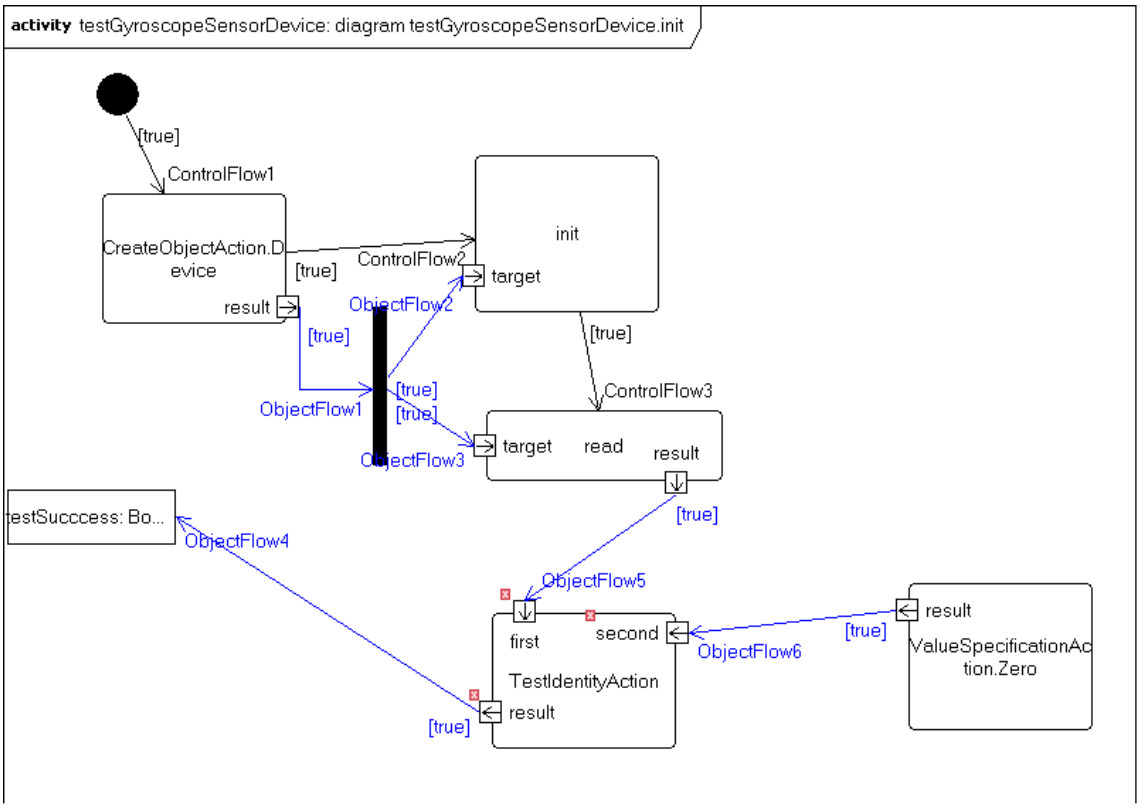

Figura A.11 - Modelagem de comportamento de um teste para a classe *GyroscopeSensorDevice*

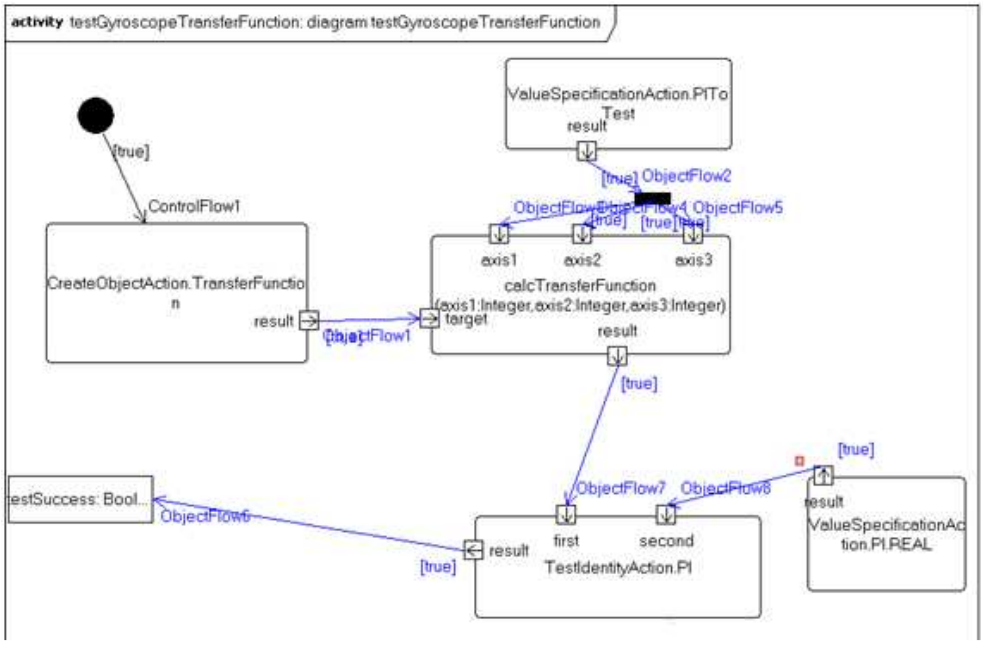

Figura A.12 - Modelagem de comportamento de um teste para a classe *GyroscopeTransferFunction*

A.3 PDM – Software - Java

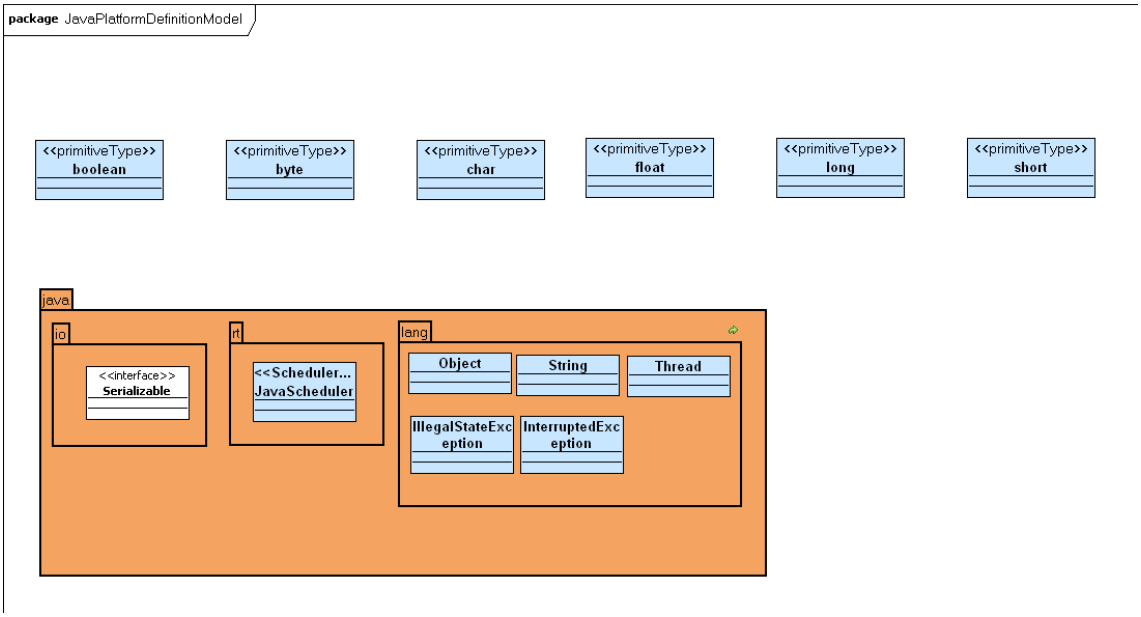

Figura A.13 - Modelagem da estrutura do modelo PDM – Software para a plataforma Java

A.4 PDM – Software – Java – Comunicação entre objetos

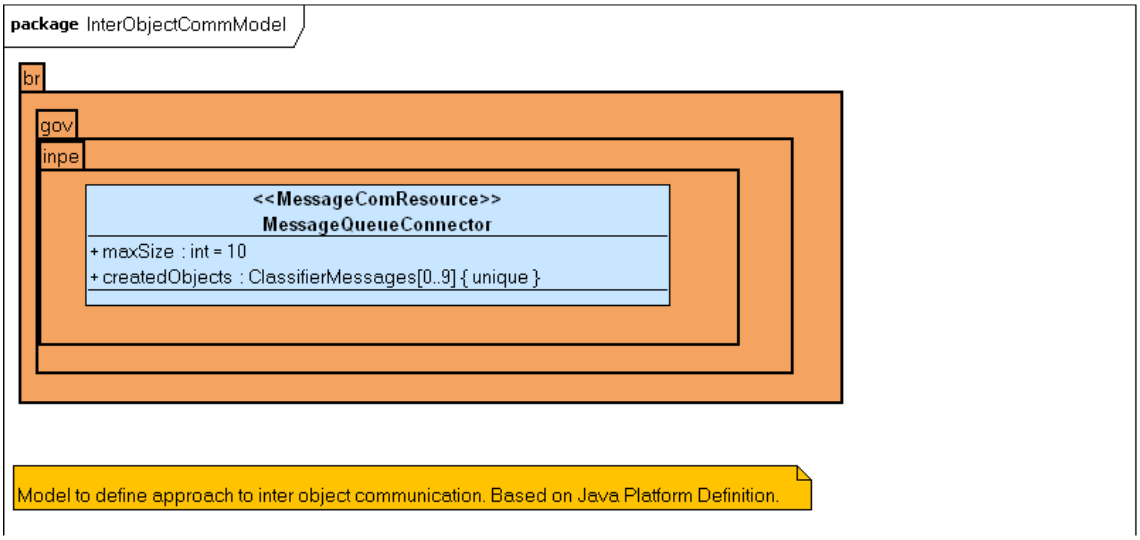

Figura A.14 - Modelagem da estrutura do modelo PDM – Software – Comunicação entre objetos para a plataforma Java

A.5 PDM – Hardware

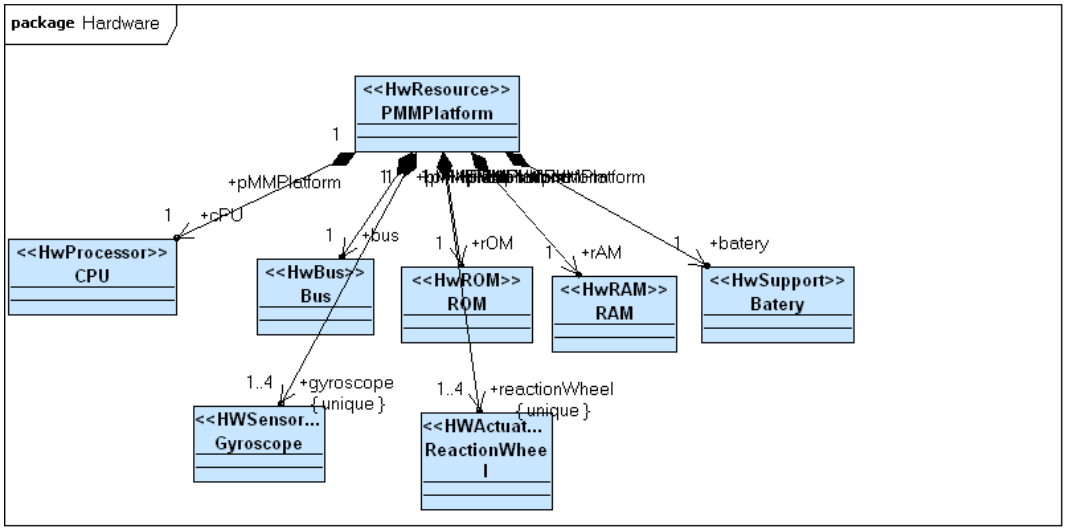

Figura A.15 - Modelagem da estrutura do modelo PDM – Hardware da PMM

A.6 PSM

Figura A.16 - Visão da estrutura gerada automaticamente para o modelo PSM

A.7 AADL

```
-- AADL
-- transformed from
PSM_Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel
_PMM_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_Sens
orModel_ClockLibrary

system
PSM_Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel
_PMM_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_Sens
orModel_ClockLibrary
end
PSM_Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel
_PMM_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_Sens
orModel_ClockLibrary;

system implementation
PSM_Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel
_PMM_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_Sens
orModel_ClockLibrary.Impl
      subcomponents
            instancied_CPU : processor CPU.Impl;
            instancied_GeneratedForAloneActivitiesImpl : process
GeneratedForAloneActivitiesImpl.Impl;
      properties
            Actual_Processor_Binding => reference instancied_CPU
applies to instancied_GeneratedForAloneActivitiesImpl;
end
PSM_Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel
_PMM_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_Sens
orModel_ClockLibrary.Impl;


processor CPU
      features
            bus_connected : requires bus access ;
end CPU;
processor implementation CPU.Impl
      properties
            Scheduling_Protocol => RATE_MONOTONIC_PROTOCOL;
end CPU.Impl;


process GeneratedForAloneActivitiesImpl
end GeneratedForAloneActivitiesImpl;
process implementation GeneratedForAloneActivitiesImpl.Impl
      subcomponents
            instancied_ControlReceiverImpl : thread
ControlReceiverImpl.Impl;
            instancied_ControlSensorRTEntityImpl : thread
ControlSensorRTEntityImpl.Impl;
            instancied_GyroscopeSensorDevice : data
GyroscopeSensorDevice.Impl;
```

```
            instancied_MessageQueueConnector : data
MessageQueueConnector.Impl;
            instancied_GyroscopeTransferFunction : data
GyroscopeTransferFunction.Impl;

      connections
            data access instancied_MessageQueueConnector ->
instancied_ControlReceiverImpl.MessageQueueConnector_features;
            data access instancied_MessageQueueConnector ->
instancied_ControlSensorRTEntityImpl.MessageQueueConnector_features;
            data access instancied_GyroscopeTransferFunction ->
instancied_ControlSensorRTEntityImpl.GyroscopeTransferFunction_feature
s;
            data access instancied_GyroscopeSensorDevice ->
instancied_ControlSensorRTEntityImpl.GyroscopeSensorDevice_features;

end GeneratedForAloneActivitiesImpl.Impl;


thread ControlSensorRTEntityImpl
      features
            MessageQueueConnector_features : requires data access
MessageQueueConnector.Impl;
            GyroscopeTransferFunction_features : requires data access
GyroscopeTransferFunction.Impl;
            GyroscopeSensorDevice_features : requires data access
GyroscopeSensorDevice.Impl;
end ControlSensorRTEntityImpl;

thread implementation ControlSensorRTEntityImpl.Impl
      properties
            Dispatch_Protocol => Periodic;
            Compute_Execution_Time =>  10 ms ..  10 ms;
            Deadline  =>  100 ms;
            Period =>  100 ms;
            Cheddar_Properties::Dispatch_Jitter => 0 ms;
end ControlSensorRTEntityImpl.Impl;


thread ControlReceiverImpl
      features
            MessageQueueConnector_features : requires data access
MessageQueueConnector.Impl;
end ControlReceiverImpl;

thread implementation ControlReceiverImpl.Impl
      properties
            Dispatch_Protocol => Periodic;
            Compute_Execution_Time =>  15 ms ..  15 ms;
            Deadline  =>  100 ms;
            Period =>  100 ms;
            Cheddar_Properties::Dispatch_Jitter => 0 ms;
end ControlReceiverImpl.Impl;

data MessageQueueConnector
end MessageQueueConnector;
```

```
data implementation MessageQueueConnector.Impl
end MessageQueueConnector.Impl;

data GyroscopeSensorDevice
end GyroscopeSensorDevice;
data implementation GyroscopeSensorDevice.Impl
end GyroscopeSensorDevice.Impl;

data GyroscopeTransferFunction
end GyroscopeTransferFunction;
data implementation GyroscopeTransferFunction.Impl
end GyroscopeTransferFunction.Impl;
```

A.8 Parte do código gerado – SensorRTEntity

```
package br.gov.inpe.physicaldevices.sensor;

import java.lang.Thread;
import
br.gov.inpe.physicaldevices.sensor.constraints.GyroscopeTransferFuncti
on;
import br.gov.inpe.physicaldevices.sensor.signals.SensorRTImageTO;
import
br.gov.inpe.physicaldevices.sensor.device.GyroscopeSensorDevice;
import br.gov.inpe.MessageQueueConnector;
import br.gov.inpe.physicaldevices.sensor.signals.TimedSignalTO;

// Start of user code for imports
// NOT RECOMMENDED - CAN BE implemented
// End of user code

/**
 *
 * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#45"
 */
public class ControlSensorRTEntityImpl extends Thread

{

      /**
       * the currentState attribute.
       *
       * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#35"
       */
      private SensorRTEntityState currentState;

      /**
       * the currentState getter.
       *
       * @return the currentState.
       */
      public SensorRTEntityState getCurrentState() {

            return this.currentState;
      }

      /**
       * the currentState setter.
       *
       * @param p_currentState the currentState to set.
       */
```

```
	public void setCurrentState(SensorRTEntityState p_currentState)
{

		this.currentState = p_currentState;
	}

	/**
	 *
	 * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#42"
	 */
	public void start() {

		// Start of user code for operation start
		// NOT RECOMMENDED - CAN BE implemented
		// End of user code
		super.start();
	}

	/**
	 *
	 * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#42"
	 */
	public void run() {

		// Start of user code for operation run
		// NOT RECOMMENDED - CAN BE implemented
		// End of user code

		// ActivityName:
		//
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior
		// NodeType: class
org.eclipse.uml2.uml.internal.impl.InitialNodeImpl
		// NodeName: InitialNode.SensorActivity
		// nothing to initial node

		// ActivityName:
		//
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior
		// NodeType: class
		//
org.eclipse.uml2.uml.internal.impl.StructuredActivityNodeImpl
		// NodeName: initialize
```

```
            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
org.eclipse.uml2.uml.internal.impl.InitialNodeImpl
            // NodeName: InitialNode1
            // nothing to initial node

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
            //
org.eclipse.uml2.uml.internal.impl.CreateObjectActionImpl
            // NodeName: CreateObjectAction.Device1
            GyroscopeSensorDevice createObjectAction_Device1_result =
new GyroscopeSensorDevice();

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
org.eclipse.uml2.uml.internal.impl.ForkNodeImpl
            // NodeName: ForkNode.Device1
            GyroscopeSensorDevice initialize_ForkNode_Device1 =
createObjectAction_Device1_result;

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
            //
org.eclipse.uml2.uml.internal.impl.CallOperationActionImpl
            // NodeName: CallOperationAction.Device1.init
            initialize_ForkNode_Device1.init();

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
```

```
            //
org.eclipse.uml2.uml.internal.impl.CreateObjectActionImpl
            // NodeName: CreateObjectAction.Device2
            GyroscopeSensorDevice createObjectAction_Device2_result =
new GyroscopeSensorDevice();

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
org.eclipse.uml2.uml.internal.impl.ForkNodeImpl
            // NodeName: ForkNode.Device2
            GyroscopeSensorDevice initialize_ForkNode_Device2 =
createObjectAction_Device2_result;

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
            //
org.eclipse.uml2.uml.internal.impl.CallOperationActionImpl
            // NodeName: CallOperationAction.Device2.init
            initialize_ForkNode_Device2.init();

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
            //
org.eclipse.uml2.uml.internal.impl.CreateObjectActionImpl
            // NodeName: CreateObjectAction.Device3
            GyroscopeSensorDevice createObjectAction_Device3_result =
new GyroscopeSensorDevice();

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
org.eclipse.uml2.uml.internal.impl.ForkNodeImpl
            // NodeName: ForkNode.Device3
            GyroscopeSensorDevice initialize_ForkNode_Device3 =
createObjectAction_Device3_result;

            // StructuredActivityNode:
```

```
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
            //
org.eclipse.uml2.uml.internal.impl.CallOperationActionImpl
            // NodeName: CallOperationAction.Device3
            initialize_ForkNode_Device3.init();

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
            //
org.eclipse.uml2.uml.internal.impl.CreateObjectActionImpl
            // NodeName: CreateObjectAction.TransferFunction
            GyroscopeTransferFunction
createObjectAction_TransferFunction_result = new
GyroscopeTransferFunction();

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
org.eclipse.uml2.uml.internal.impl.ForkNodeImpl
            // NodeName: ForkNode4
            GyroscopeTransferFunction initialize_ForkNode4 =
createObjectAction_TransferFunction_result;

            // StructuredActivityNode:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::initialize
            // NodeType: class
            // org.eclipse.uml2.uml.internal.impl.ActivityFinalNodeImpl
            // NodeName: ActivityFinalNode1
            // nothing to final node

            // ActivityName:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior
            // NodeType: class
org.eclipse.uml2.uml.internal.impl.MergeNodeImpl
```

```
            // NodeName: MergeNode1
            do {

                  // ActivityName:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior
                  // NodeType: class
                  //
org.eclipse.uml2.uml.internal.impl.AcceptEventActionImpl
                  // NodeName: AcceptEventAction.TimedSignal

                  // StructuredActivityNode:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                  // NodeType: class
                  //
org.eclipse.uml2.uml.internal.impl.ReadStructuralFeatureActionImpl
                  // NodeName: ReadStructuralFeatureAction.TimedSignal
                  double readStructuralFeatureAction_TimedSignal_result
= System.currentTimeMillis();

                  // ActivityName:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior
                  // NodeType: class
                  //
org.eclipse.uml2.uml.internal.impl.StructuredActivityNodeImpl
                  // NodeName: reading

                  // StructuredActivityNode:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                  // NodeType: class
                  // org.eclipse.uml2.uml.internal.impl.InitialNodeImpl
                  // NodeName: InitialNode1
                  // nothing to initial node

                  // StructuredActivityNode:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                  // NodeType: class
```

```
                    //
org.eclipse.uml2.uml.internal.impl.CallOperationActionImpl
                    // NodeName: CallOperationAction.Device1.read
                    double callOperationAction_Device1_read_readValue =
initialize_ForkNode_Device1.read();

                    // StructuredActivityNode:
                    //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                    // NodeType: class
                    //
org.eclipse.uml2.uml.internal.impl.CallOperationActionImpl
                    // NodeName: CallOperationAction.Device2.read
                    double callOperationAction_Device2_read_result =
initialize_ForkNode_Device2.read();

                    // StructuredActivityNode:
                    //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                    // NodeType: class
                    //
org.eclipse.uml2.uml.internal.impl.CallOperationActionImpl
                    // NodeName: CallOperationAction.Device3.read
                    double callOperationAction_Device3_read_result =
initialize_ForkNode_Device3.read();

                    // StructuredActivityNode:
                    //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                    // NodeType: class
                    //
org.eclipse.uml2.uml.internal.impl.CallOperationActionImpl
                    // NodeName: CallOperationAction.TransferFunction
                    double[]
callOperationAction_TransferFunction_calcValue =
initialize_ForkNode4.calcTransferFunction(callOperationAction_Device1_
read_readValue, callOperationAction_Device2_read_result,
callOperationAction_Device3_read_result);

                    // StructuredActivityNode:
                    //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                    // NodeType: class
```

```
                  //
org.eclipse.uml2.uml.internal.impl.ReadExtentActionImpl
                  // NodeName: ReadExtentAction
                  // nothing to read extend because messagequeue

                  // StructuredActivityNode:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                  // NodeType: class
                  //
org.eclipse.uml2.uml.internal.impl.SendSignalActionImpl
                  // NodeName: SendSignalAction.Sensor
                  // creating signal
                  SensorRTImageTO sendSignalAction_Sensor_value = new
SensorRTImageTO();

      sendSignalAction_Sensor_value.setValue(callOperationAction_Trans
ferFunction_calcValue);

      sendSignalAction_Sensor_value.setTime(readStructuralFeatureActio
n_TimedSignal_result);

      MessageQueueConnector.getInstance().send("ControlReceiverImpl",
sendSignalAction_Sensor_value);

                  // StructuredActivityNode:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior::reading
                  // NodeType: class
                  //
org.eclipse.uml2.uml.internal.impl.ActivityFinalNodeImpl
                  // NodeName: ActivityFinalNode1
                  // nothing to final node

                  // ActivityName:
                  //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior
                  // NodeType: class
                  //
org.eclipse.uml2.uml.internal.impl.DecisionNodeImpl
                  // NodeName: DecisionNode1
                  // occKind periodic (period=(100,ms), jitter=(0,us))
                  try {
                        Thread.sleep(100);
                  } catch (java.lang.InterruptedException ie) {
                        // nothing
                  }
```

```
            } while (true);

            // ActivityName:
            //
Woven_Woven_Woven_InterObjectCommModel_JavaPlatformDefinitionModel_PMM
_PlatformDefinitionModel_Woven_Woven_PIM_SensorModel_PIM_Test_SensorMo
del_ClockLibrary::PhysicalDevices::Sensor::SensorRTEntity::PhysicalDev
ices.ClassifierBehavior
            // NodeType: class
            // org.eclipse.uml2.uml.internal.impl.ActivityFinalNodeImpl
            // NodeName: ActivityFinalNode.SensorActivity
            // nothing to final node

      }

      /**
       *
       * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#42"
       */
      public void processEvent() {

            // Start of user code for operation processEvent
            // NOT RECOMMENDED - CAN BE implemented
            // End of user code
      }

      /**
       * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#48"
       */
      public interface SensorRTEntityState {
      }

      /**
       * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#45"
       */
      public class Initialize

      implements SensorRTEntityState {

      }

      /**
       * @mygenerated
"sourceid:org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl@9444d1
uri='file:/C:/Documents%20and%20Settings/Administrator/Desktop/Sensor/
Model/generated/PSM.uml'#45"
       */
```

```
public class Reading
implements SensorRTEntityState {}}
```

APÊNDICE B – LISTA DE PUBLICAÇÕES

FULINDI, J. B.; LOUREIRO, G.; ROMERO, A. G.; KUCINSKIS, F. N.; LEMONGE, C. E. A.; VAZQUEZ, R. F.; MIYASHIRO, M. A. S. Electrical Ground Support Equipment (EGSE) para um Onboard Computer (OBC). In: WORKSHOP EM ENGENHARIA E TECNOLOGIA ESPACIAIS, 1., 2010, São José dos Campos. Anais... 2010. DVD.

LOUREIRO, G.; FULINDI, J. B.; ROMERO, A. G.; KUCINSKIS, F. N.; LEMONGE, C. E. A.; VAZQUEZ, R. F.; MIYASHIRO, M. A. S. Systems Concurrent Engineering of an Electrical Ground Support Equipment for an On-Board Computer. In: 17TH ISPE INTERNATIONAL SOCIETY FOR PRODUCTIVITY ENHANCEMENT, 2010, Cracóvia Londres. Proceedings... 2010. Disponível em: <http://www.ce2010.pl>. Acesso em: 29 set. 2010.

Romero, Alessandro Gerlinger. Ferreira, Mauricio Gonçalves Vieira. Using Semantic of Foundational for Executable UML Models to apply Model-Driven Architecture on real time systems. **Submetido** em 28 set. 2010 para a revista “Design Automation for Embedded Systems”.

Romero, Alessandro Gerlinger. Ferreira, Mauricio Gonçalves Vieira. Modeling and Verifying Real Time Systems. **Submetido** em 23 set. 2010 para o simpósio “Fifth Latin-American Symposium on Dependable Computing”.

Romero, Alessandro Gerlinger. Ferreira, Mauricio Gonçalves Vieira. Platform Independence and Fault Tolerance. **Submetido** em 23 set. 2010 para o simpósio “Fifth Latin-American Symposium on Dependable Computing”.