UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
E INFORMÁTICA INDUSTRIAL

LUCIANE TELINSKI WIEDERMANN AGNER

PI-MT: MÉTODO PARA A CRIAÇÃO DE TRANSFORMAÇÕES DE MODELOS NO CONTEXTO DA MDA

TESE

CURITIBA
2012

LUCIANE TELINSKI WIEDERMANN AGNER

PI-MT: MÉTODO PARA A CRIAÇÃO DE TRANSFORMAÇÕES DE MODELOS NO CONTEXTO DA MDA

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, como requisito parcial para obtenção do título de Doutor em Ciências - Área de Concentração: Engenharia de Computação.

Orientador: Prof. Dr. Paulo Cézar Stadzisz

Co-Orientador: Prof. Dr. Jean Marcelo Simão

CURITIBA
2012

Dados Internacionais de Catalogação na Publicação

A271 Agner, Luciane Telinski Wiedermann
PI-MT : método para a criação de transformações de modelos no contexto da MDA / Luciane Telinski Wiedermann. — 2012.
182 f. : il. ; 30 cm

Orientador: Paulo Cezar Stadzisz.
Co-orientador: Jean Marcelo Simão.
Tese (Doutorado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Área de concentração: Engenharia de Computação, Curitiba, 2012.
Bibliografia: p. 168-182.

1. Arquitetura de software orientada a modelos. 2. Sistemas embutidos de computador. 3. Sistemas embarcados (Computadores). I. Stadzisz, Paulo Cezar, orient. II. Simão, Jean Marcelo, co-orient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD (22. ed.) 621.3

Biblioteca Central da UTFPR, Campus Curitiba

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Câmpus Curitiba

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

Tese de Doutorado Nº. 80

"PI-MT: Um Método para a Criação de Transformações de Modelos no Contexto da MDA."

por

Luciane Telinski Wiedermann Agner

Esta tese foi apresentada como requisito parcial à obtenção do título de Doutor em CIÊNCIAS - Área de Concentração: Engenharia de Computação, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI, da Universidade Tecnológica Federal do Paraná – UTFPR, às 14:30h do dia 14 de dezembro de 2012. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:

Prof. Paulo Cézar Stadzisz, Dr.
(Presidente)

Prof. Marcos Antônio Quináia, Dr.
(UNICENTRO)

Profª. Silvia Regina Vergílio, Dr.
(UFPR)

Prof. André Koscianski, Dr.
(UTFPR - PG)

Prof. Cesar Augusto Tacla, Dr.
(UTFPR)

Visto da Coordenação:

Prof. Ricardo Lüders, Dr.
(Coordenador do CPGEI)

Ao Julio e à Julia

AGRADECIMENTOS

Primeiramente, agradeço a Deus e à Virgem Maria, pelas inúmeras e valiosas bênçãos derramadas em minha vida.

Em especial, à minha filha e ao meu esposo, pela felicidade e amor com que preenchem minha vida. Agradeço também pela paciência, compreensão, carinho e incentivo.

Ao meu orientador, Profº. Paulo Cézar Stadzisz, pela orientação, conhecimentos e experiências transmitidos ao longo destes anos e fundamental apoio durante toda a realização deste trabalho.

Ao meu co-orientador, Profº. Jean Marcelo Simão, pela atenção, incentivo, prontidão e colaboração durante o desenvolvimento deste trabalho.

À minha mãe, pelo exemplo e incentivo.

À minha família, grande alicerce da minha vida.

Às minhas grandes amigas, Inali e Josiane, que estiveram ao meu lado durante toda a realização desta pesquisa. Obrigada pelo apoio, carinho, incentivo e amizade verdadeira.

Aos professores André Koscianski, Marcos Antônio Quináia, Silvia Regina Vergilio e Cesar Augusto Tacla pela participação como membros da banca de avaliação desta tese.

A todos meus amigos e colegas da Unicentro que me apoiaram durante a realização desta pesquisa.

Aos funcionários e professores da UTFPR por todo auxílio prestado, em especial à Terezinha.

À Fernanda e à Duda pelo carinho e amizade.

“*Mãe, nada sem vós, nada sem vós.*
Maria passa na frente!”

RESUMO

AGNER, Luciane Telinski Wiedermann. PI-MT: Um Método para a Criação de Transformação de Modelos no Contexto da MDA. 2012. 182 f. Tese de Doutorado - Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI), Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba, 2012.

Esta tese aborda o problema de prover a independência de plataforma em transformações de modelos no contexto da MDA (*Model Driven Architecture*). A MDA é uma abordagem de desenvolvimento de *software* baseada na criação e na transformação de modelos. Assim, em MDA os modelos são os principais artefatos ao longo do ciclo de vida de desenvolvimento de *software* e as transformações desempenham um papel fundamental. No entanto, a maioria das iniciativas MDA são dedicadas, isto é, as características da plataforma são implicitamente especificadas nas regras de transformação. À medida que os aspectos ligados à plataforma não são separados das regras de transformação, para cada plataforma selecionada deve haver uma transformação de modelos correspondente. Por um lado, isto facilita o desenvolvimento de transformações de modelo. Por outro lado, a transformação torna-se limitada a uma plataforma específica, uma vez que os recursos da plataforma estão fortemente associados com as regras de transformação. Um dos desafios consiste em criar regras de transformação independentes das características da plataforma de implementação de sistemas de *software*. Esta tese propõe um método para a criação de transformações de modelo chamado PI-MT (*Platform Independent - Model Transformations*). O PI-MT é voltado ao desenvolvimento de *software* embarcado baseado em Sistemas Operacionais em Tempo-Real (RTOS - *Real-Time Operating System*). Além disso, o método permite a adaptação de transformações de modelos para diferentes plataformas baseadas em RTOS, por meio de modelos de plataforma explicitamente definidos. Como resultado, o método de PI-MT oferece independência entre as regras de transformação e as características da plataforma.

Palavras-chave: *Model Driven Architecture* (MDA). Transformação de Modelos. Refinamento de Modelos. Modelo de Plataforma. *Software* Embarcado.

ABSTRACT

AGNER, Luciane Telinski Wiedermann. PI-MT: A Method for the Design of Model Transformations in the Context of MDA. 2012. 182 f. Thesis - Graduate School of Electrical Engineering and Computer Science (CPGEI), Paraná Federal University of Technology (UTFPR). Curitiba, 2012.

This thesis deals with the platform-independence matter in the Model Driven Architecture (MDA) context. MDA is an approach to software development based on the design and transformation of models. In the MDA approach models are the core artifacts throughout the entire software development lifecycle, and thus a key role is played by model transformations. Nevertheless, most MDA initiatives are dedicated, i.e., the platform features are implicitly employed in the transformation rules. As the aspects associated with the platform are not separated from the transformation rules, for each selected platform there must be a corresponding model transformation. On the one hand, this makes the model transformation development easier. On the other hand, model transformation becomes limited to a specific platform, once the platform features are strongly associated with the transformation rules. An open challenge consists of how to create transformation rules independently of the deployment platform features of software systems. This thesis proposes a method for creating model transformations, called PI-MT (Platform Independent - Model Transformations). The PI-MT is particularly applied to embedded software development based on Real-Time Operating Systems (RTOS). In addition, it allows the adaptation of the model transformation process to different RTOS-based platforms by means of Platform Models explicitly defined. As a result, the PI-MT method provides independence between the model transformation rules and the platform features.

Keywords: *Model Driven Architecture* (MDA). Models Transformation. Models Refinement. Platform Model. Embedded Software.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

AMP	Application Modeling Profile
API	Application Programming Interface
ARM	Advanced RISC Machines
ATL	Atlas Transformation Language
CISC	Complex Instruction Set Computer
CWM	Common Warehouse Metamodel
EMF	Eclipse Modeling Frameworks
GEF	Graphical Editing Framework
IDE	Integrated Development Environment
INRIA	Institut National de Recherche en Informatique et en Automatique
LCD	Liquid Crystal Display
LHS	Left Hand Side
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MOF	Meta-Object Facility
MT	Model Transformation
MTL	Model Transformation Language
M2M	Model to Model
OCL	Object Constraint Language
OMG	Object Management Group
PCB	Printed Circuit Board
PIM	Platform Independent Model
PI-MT	Platform Independent – Model Transformation
PM	Platform Model
POSIX	Portable Operating System Interface
PROAPES	Profile for modeling Application and Platform of Embedded Software
PSM	Platform Specific Model
QVT	Queries / Views / Transformations (QVT)
RCP	Rich Client Platform
RHS	Right Hand Side
RISC	Reduced Instruction Set Computer
RTOS	Real-Time Operating Systems
SA	Simulador de Alarme
SDK	Software Development Kit
SysML	Systems Modeling Language
SoC	System-on-a-Chip
SOM	System-on-Module
TOPCASED	Toolkit in OPen-source for Critical Application & SystEms Development
UML	Unified Modeling Language
VHDL	VHSIC - Very High Speed Integrated Circuits - Hardware Description Language
VLIW	Very Long Instruction Word
XMI	XML Metadata Interchange

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta o contexto deste trabalho de pesquisa, bem como seus objetivos e sua relevância. O método empregado no desenvolvimento desta tese também é descrito. Por fim, a organização dos capítulos que compõe este documento é apresentada.

1.1 CONTEXTO DO TRABALHO DE PESQUISA

Atualmente, sistemas de *software* são amplamente utilizados e estão presentes em equipamentos e aplicações destinados aos mais diversos fins. Em adição, a crescente demanda por novos produtos exige que pessoas desenvolvam *software* que façam mais coisas, de forma mais rápida e segura, o que resulta em um aumento da complexidade. Consequentemente, torna-se mais difícil garantir a qualidade do *software* produzido. Além disso, para desenvolver artefatos de *software* mais complexos, a produtividade é afetada.

A Engenharia de *Software* (ES) é uma disciplina de engenharia que se ocupa de todos os aspectos da produção de *software* (SOMMERVILLE, 2003). Particularmente, dois elementos da Engenharia de *Software* que podem trazer melhorias significativas em termos de qualidade e produtividade são a modelagem e a automação. A modelagem trata de abstração e da concepção do *software*, além de envolver o uso de linguagens padronizadas, como a *Unified Modeling Language* (UML). Em adição, a modelagem é impactante em ES, pois facilita a comunicação entre os desenvolvedores, permite a análise do *software* a partir de diferentes perspectivas, possibilita especificar a estrutura e o comportamento do *software* de forma compreensível, além de auxiliar na geração da documentação (GUEDES, 2009; SOMMERVILLE, 2003).

Embora não seja mandatória, em abordagens de desenvolvimento tradicionais a UML é frequentemente utilizada. Assim, quando a modelagem é utilizada em abordagens tradicionais, os modelos de *software* são construídos e, posteriormente, programas são codificados com base nesses modelos. Desse modo, tarefas redundantes são realizadas, visto que modelo e código são definidos em etapas distintas do ciclo de desenvolvimento de *software*, mas representam, por meio de diferentes linguagens, o mesmo sistema de *software*.

Por sua vez, a automação da geração de código é um importante elemento que pode contribuir para aumentar a produtividade e melhorar a qualidade, por tornar o

desenvolvimento menos propenso a erros e reduzir o esforço exigido dos desenvolvedores. Quando o código é gerado automaticamente (ou semi-automaticamente) a partir da modelagem do *software*, a probabilidade de ocorrência de erros é menor, pois tarefas redundantes são evitadas, o que resulta em maior qualidade do *software* produzido (MELLOR *et al*., 2005; HERRINGTON, 2003). Além disso, nesse caso a etapa de codificação do *software* é realizada de forma automatizada (com base na modelagem), aumentando, assim, a produtividade (HERRINGTON, 2003).

Existe um tipo de sistema em especial, chamado de sistema embarcado, que são sistemas compostos de *hardware* e *software* e projetados para executar uma função dedicada (WOLF, 2012). O mercado de sistemas embarcados tem apresentado um grande potencial de expansão (EBERT e JONES, 2009) e um crescimento muito superior se comparado ao mercado de *Personal Computers* (PCs). É importante ressaltar que, no desenvolvimento de *software* embarcado, garantir a qualidade e incrementar a produtividade são desafios ainda maior, devido à grande variedade de plataformas embarcadas existentes e à complexidade inerente desse tipo de *software*. A natureza de plataformas embarcadas é diferente se comparada com arquiteturas de PCs. Por exemplo, dado que plataformas embarcadas consistem da combinação de *hardware* e sistemas operacionais especializados, elas resultam em milhares de plataformas possíveis de serem utilizadas. Em adição, isto faz com que a portabilidade se torne ainda mais importante em sistemas embarcados.

Sistemas embarcados geralmente fazem uso de Sistemas Operacionais em Tempo Real (*Real-Time Operating System* - RTOS) para suporte à sua construção e execução. Características específicas diferenciam um RTOS de um sistema operacional de propósito geral, como serviço de interrupção com limite máximo do tempo de resposta, tarefas preemptivas e escalabilidade (MARWEDEL, 2006). Em adição, sistemas embarcados possuem recursos limitados e necessitam de sistemas operacionais adaptados à plataforma de *hardware* e ao domínio da aplicação.

Com base no que foi mostrado, é pertinente salientar que novas abordagens de Engenharia de *Software* têm apresentado soluções visando à melhoria do produto de *software* em geral, sendo que essas soluções poderiam (ou até mesmo deveriam) ser consideradas para a questão de *software* embarcado. Neste contexto, durante a última década, surgiu um novo conceito em Engenharia de *Software*, denominado “abordagens dirigidas a modelos”. Essas abordagens, referenciadas pelos termos: *Model Driven Engineering* (MDE), *Model Driven Architecture* (MDA) e *Model Driven Development* (MDD), enfatizam o uso da modelagem de

software e surgiram para permitir a automação (total ou parcial) da geração de código-fonte (GHERBI *et al*., 2009).

Abordagens dirigidas a modelos promovem o uso de modelos como principais artefatos em todas as fases de desenvolvimento de *software*: especificação do sistema, projeto, implementação e testes (FRANCE e RUMPE, 2007). Esses modelos podem estar no mesmo nível de abstração ou em níveis de abstrações diferentes. Modelos mais abstratos estão mais distantes das particularidades de uma determinada plataforma, enquanto modelos menos abstratos estão mais próximos das especificações dessa plataforma (SELIC, 2003). Manter os artefatos de *software* na forma de modelos torna esses artefatos mais flexíveis ou adaptáveis às mudanças e evolução de tecnologias. Desse modo, a necessidade de modelagem de *software* é enfatizada, sendo o código-fonte do *software* gerado (total ou parcialmente) a partir de seus modelos, por meio da aplicação de transformações.

Dentre essas abordagens, a MDA, proposta pelo *Object Management Group* (OMG), é a mais conhecida e adotada atualmente (DUBE e DIXIT, 2012; AMELLER *et al*., 2010; SINGH e SOOD, 2009). Nessa tese, será usado com maior frequência o termo MDA devido ao referencial conceitual e tecnológico fornecido pelo OMG. Entretanto, as proposições desta tese aplicam-se, da mesma forma, à MDE e à MDD.

No ciclo de desenvolvimento de *software* em MDA, um Modelo Independente de Plataforma (*Platform Independent Model* - PIM) é transformado em um Modelo Específico de Plataforma (*Platform specific Model* - PSM), com base em um Modelo de Plataforma (*Platform Model* - PM). PIMs representam a estrutura e as funcionalidades do sistema de *software* e são desenvolvidos com o auxílio de uma linguagem de modelagem, como a UML. Por sua vez, o modelo PSM define detalhes de uma plataforma específica e é o refinamento do modelo PIM. Finalmente, o PSM é transformado em código-fonte do *software* (OMG, 2003). O termo “plataforma” significa um conjunto de mecanismos de *hardware* e *software* que suportam a execução de aplicações de *software* (SELIC, 2005). Por sua vez, um Modelo de Plataforma (PM), provê um conjunto de conceitos técnicos, representando as partes e os serviços de uma determinada plataforma.

A transformação de modelos (MT - *Model Transformation*) desempenha um papel fundamental em MDA, sendo muitas vezes referenciada como o “coração” e a “alma” da MDA (LAFI *et al*., 2011; LOUHICHI *et al*., 2011; DEKEYSER *et al*., 2005; LOPES *et al*., 2005). Transformações de modelos em MDA são baseadas em mapeamentos. Um mapeamento é formado por regras que definem uma correspondência entre os elementos do modelo-fonte e os elementos do modelo-alvo. Desse modo, em MDA um mapeamento provê

especificações para o processo de transformação de um PIM em um PSM com base em uma determinada plataforma (SENDALL e KOZACZYNSKI, 2003; OMG, 2003).

No entanto, a maioria das abordagens de transformação MDA são dedicadas, ou seja, todos os detalhes da plataforma são incorporados juntamente com as regras de transformação (ANASTASAKIS *et al*., 2010; SUN *et al*. 2009; LOPES *et al*., 2005). Dessa forma, as regras de transformação incluem uma mistura de aspectos da aplicação e de aspectos específicos da plataforma de implementação. Em consequência, portar o *software* para outra plataforma requer reescrever as regras de transformação para incorporar as características da nova plataforma. Como resultado, transformações de modelos dedicadas dificultam a portabilidade do *software* para novas plataformas (TRATT, 2005; WAGELAAR e JONCKERS, 2005).

Na verdade, transformações de modelos dedicadas meramente deslocam o problema de oferecer "indepêndencia de plataforma" do desenvolvimento da aplicação para o desenvolvimento da transformação de modelos. Em sistemas embarcados este problema se agrava ainda mais, visto que esses sistemas podem ser implementados utilizando-se diversas plataformas, cada qual com características e restrições específicas.

Em MDA, a UML tem uso intensivo por meio do emprego de modelos, diferentemente do uso ocasional ou voluntário feito da UML em outras abordagens de desenvolvimento tradicionais. O conceito de modelagem não é novo, entretanto, a novidade está no uso intensivo da modelagem como objeto central no desenvolvimento de *software* dirigido a modelos. Como consequência, os desenvolvedores passam a ter um foco maior em representações mais abstratas do *software*. Desse modo, um projeto da arquitetura do *software* é construído por meio da modelagem UML, permitindo assim a melhoria da capacidade de compreensão e a ênfase na arquitetura do *software*.

Os principais benefícios esperados pelo uso da MDA são a melhoria da qualidade do *software* e da produtividade no processo de desenvolvimento. Pesquisas realizadas apontam que o incremento da produtividade é obtido em MDA, principalmente, por meio da simplificação do processo de desenvolvimento, da melhoria da comunicação entre os membros da equipe e da maior facilidade na geração de código (AGNERc *et al*., 2012; BASHA *et al*., 2012; NUGROHO e CHAUDRON, 2008; MOHAGHEGHI E DEHLEN; 2008). Por sua vez, a melhoria da qualidade de *software* é percebida, principalmente, na redução do número de defeitos no *software* obtida por meio do uso de modelos com alto nível de abstração e por diminuir a realização de tarefas redundantes (MOHAGHEGHI E DEHLEN, 2008). Além disso, a qualidade do *software* desenvolvido também é maior, pois sua arquitetura é explicitamente concebida.

Um estudo realizado durante o desenvolvimento dessa tese, constatou que no Brasil a maioria das empresas desenvolvedoras de *software* embarcado é iniciante em modelagem de *software* (em torno de 45%), bem como no desenvolvimento de *software* dirigido a modelos (em torno de 10%) (AGNERc *et al*., 2012). A falta de profissionais com habilidades específicas em UML e o curto prazo para o desenvolvimento do *software* foram os principais motivos citados como causa. Resultados semelhantes foram obtidos por outras pesquisas que buscaram identificar o nível de uso da UML e da MDA no desenvolvimento de *software* de propósito geral, não considerando sistemas embarcados em específico (NUGROHO e CHAUDRON, 2008; MOHAGHEGHI e DEHLEN, 2008; HUTCHINSON *et al*., 2011). Este cenário deverá mudar em um futuro próximo, visto que estas áreas (modelagem e MDA) têm recebido grande atenção da comunidade científica e empresarial (BASHA *et al*., 2012; DUBE e DIXIT, 2012), sendo alvo de importantes pesquisas.

1.2 OBJETIVOS

O objetivo principal desta tese é propor um método para a criação de transformações de modelos reutilizáveis no desenvolvimento de *software* embarcado baseado em Sistemas Operacionais em Tempo Real, denominado PI-MT (*Platform Independent - Model Transformations*). A partir desse objetivo principal, foram estabelecidos os seguintes objetivos específicos:

- Buscar identificar o nível de uso de modelagem UML e de abordagens dirigidas a modelo na indústria de *software* embarcado no Brasil;
- Estudar as principais classificações de transformações de modelos existentes atualmente na literatura;
- Avaliar as principais técnicas de refinamento existentes, visto que as transformações de modelos no contexto da MDA referem-se ao refinamento de modelos de *software*;
- Estudar e detalhar os elementos para compor o método proposto;
- Avaliar e selecionar uma linguagem de transformação de modelos adequada para a implementação das transformações de modelos propostas, bem como, avaliar as limitações e dificuldades da linguagem de transformação selecionada;

- Propor e implementar transformações de modelos baseadas no método proposto, voltadas para a modelagem estrutural e comportamental de sistemas embarcados baseados em RTOS;
- Selecionar uma técnica de verificação e testes de transformações de modelos para realizar os testes das transformações implementadas nesta pesquisa;
- Desenvolver um caso de estudo que implemente um sistema embarcado baseado em RTOS.

1.3 RELEVÂNCIA E MOTIVAÇÃO

Alguns poucos estudos têm ressaltado a importância da integração de abordagens dirigidas a modelos com o processo de desenvolvimento de *software* embarcado, a fim de minimizar os efeitos da heterogeneidade das plataformas e a inerente complexidade desse tipo de *software* (GERARD *et al*., 2010; JEON *et al*., 2009; KARSAI *et al*., 2008). Esta situação contrasta com a dimensão do mercado de sistemas embarcados que são amplamente utilizados em vários tipos de dispositivos, tais como: produtos industriais, produtos de consumo e equipamentos de defesa.

Em adição, a grande demanda por novos produtos embarcados com funcionalidades adicionais é uma tendência, exigindo, assim, o incremento de componentes de *software* mais complexos. Dessa forma, a crescente complexidade dos sistemas de *software* embarcados enfatiza a necessidade de abordagens de desenvolvimento que ofereçam maior produtividade e qualidade, como a MDA (KARSAI *et al*., 2008).

Do mesmo modo, *software* embarcado geralmente é objeto de rígidas restrições, pois é desenvolvido para uma plataforma específica (i.e., uma combinação particular de *hardware* e *software* básico). Isso dificulta o reuso do *software* desenvolvido em diferentes plataformas, devido à personalização necessária que depende da plataforma adotada. Esses fatores representam um grande desafio para a comunidade de desenvolvedores de *software* embarcado (MARWEDEL, 2006). Segundo Espinoza *et al*. (2009), o uso de abordagens dirigidas a modelos na concepção de *software* embarcado beneficia o reuso e a evolução de *software*.

Com a evolução dos sistemas embarcados, que têm aumentado o número de suas funções e a sua complexidade, tem havido um aumento crescente no emprego de sistemas operacionais embarcados. Esses sistemas são frequentemente denominados Sistemas

Operacionais em Tempo-Real (*Real-Time Operating System* – RTOS) e oferecem um conjunto de serviços padronizados para o gerenciamento de recursos do *hardware* embarcado. Um RTOS diferencia-se de um sistema operacional de uso comum principalmente por oferecer suporte para execução de sistemas embarcados (DOUGLASS, 1999).

O apoio oferecido pela MDA para o desenvolvimento de *software* embarcado, principalmente para *software* baseado em RTOS, ainda é bastante limitado. Este trabalho de pesquisa tem como tema específico o desenvolvimento de *software* embarcado utilizando RTOS.

A transformação de modelos possui um papel chave em MDA. A maioria das pesquisas sobre transformação de modelos, no contexto da MDA, são dedicadas e definem os aspectos da plataforma juntamente com as regras de transformação (JEON *et al*., 2009; ANASTASAKIS *et al*., 2010; SUN *et al*. 2009; LOPES *et al*., 2005). Nesses casos, um Modelo de Plataforma (PM) explicitamente definido não é utilizado e a transformação torna-se restrita a uma plataforma específica (CZARNECKI e HELSEN, 2006; TRATT, 2006). Algumas poucas iniciativas fazem uso do conceito de PM explicitamente definido (SANDRIESER *et al*., 2011; SELIC, 2005; WAGELAAR e JONCKERS, 2005; KUKKALA *et al*., 2005). Essas iniciativas visam à construção e uso de Modelos de Plataforma, entretanto, não fornecem artefatos específicos para modelar serviços de sistemas embarcados baseados em RTOS.

Em transformações de modelos genéricas, para cada plataforma-alvo utilizada deve existir uma transformação de modelos configurada para essa plataforma. O método proposto nesta tese se destaca por possibilitar a criação de transformações genéricas, aplicáveis a novas plataformas, por meio do uso de PMs explicitamente definidos e da definição de regras de transformação independentes da plataforma adotada. Assim, o trabalho realizado nessa tese é relevante, pois propõe um método para desenvolver transformações MDA que oferecem a melhoria no reuso e portabilidade dos modelos de *software*, de forma sistemática.

Um *framework* para transformações de modelos baseado no perfil *Modeling and Analysis of Real-Time and Embedded Systems* (MARTE) é proposto por Chehade *et al*. (2011). Essa pesquisa tem como foco a concepção de transformações de modelos genéricas, voltadas para aplicações baseadas em Sistemas Operacionais em Tempo Real. O trabalho de pesquisa envolvendo o MARTE foi a única referência efetiva encontrada no tema desta tese.

O perfil MARTE define elementos que permitem a modelagem de sistemas embarcados em tempo real baseados em diferentes RTOSs (GERARD *et al*., 2007). É pertinente ressaltar que, plataformas embarcadas baseadas em RTOS possuem APIs

(*Application Program Interfaces*) e padrões de implementação que podem variar significativamente, o que requer um escopo muito amplo a ser coberto pelo perfil MARTE.

Assim, a modelagem do sistema de *software* para um RTOS específico, usando o MARTE, requer a adaptação desse perfil às convenções de modelagem do sistema operacional selecionado, o que se caracteriza como um processo moroso e que requer um conhecimento aprofundado por parte do desenvolvedor sobre os elementos definidos nesse perfil. Nesse caso, os principais problemas encontrados são a complexidade e a generalidade de uso do MARTE, que pode ser adaptado para diferentes RTOS e que contempla em sua especificação muitos elementos. Ainda, segundo Silvestre (2012) a especificação do MARTE é redundante, complexa e oferece pouca documentação.

1.4 METODOLOGIA

A metodologia utilizada no desenvolvimento deste trabalho, adaptada de (MAFRA e BARCELOS, 2006; SHULL *et al*., 2001; LEAL, 2010) e ilustrada na Figura 1, é composta pelas seguintes etapas: Revisão Bibliográfica, Qualificação, Revisão Sistemática, Desenvolvimento da Pesquisa, Avaliação e Redação.

A primeira etapa consistiu da revisão bibliográfica de temas relacionados com esta tese. Foram estudados os principais conceitos relacionados com sistemas embarcados, dentre eles: restrições de sistemas embarcados, arquitetura de processadores, desenvolvimento de *software* embarcado e Sistemas Operacionais em Tempo Real. Os princípios que regem a abordagem MDA também fizeram parte desta etapa de revisão, sendo eles: ciclo de desenvolvimento de *software* segundo a abordagem MDA, modelos, metamodelos, transformações de modelos, padrões MDA e ferramentas MDA.

A segunda etapa contemplou os requisitos necessários para realizar a qualificação da proposta de tese e consistiu da redação do texto e da qualificação da defesa. Em seguida, realizou-se a Revisão Sistemática que englobou os conceitos específicos utilizados no desenvolvimento desta pesquisa, sendo eles: classificação de abordagens de transformação de modelos, a linguagem de transformação *Atlas Transformation Language* (ATL) e técnicas de refinamento de modelos utilizando a linguagem ATL.

A próxima etapa consistiu da definição do método proposto por esta pesquisa, denominado PI-MT, seguida pela implementação de duas transformações de modelos, chamadas de MT-AMP e MT-PROAPES, voltadas para modelagem estrutural e modelagem

comportamental de *software* embarcado, respectivamente. Ambas as transformações foram implementadas com base no método proposto e utilizando a linguagem ATL. Concorrentemente, um *survey* intitulado "Estudo da utilização de modelagem no desenvolvimento de *software* para sistemas microprocessados (*software* embarcado)" foi desenvolvido.

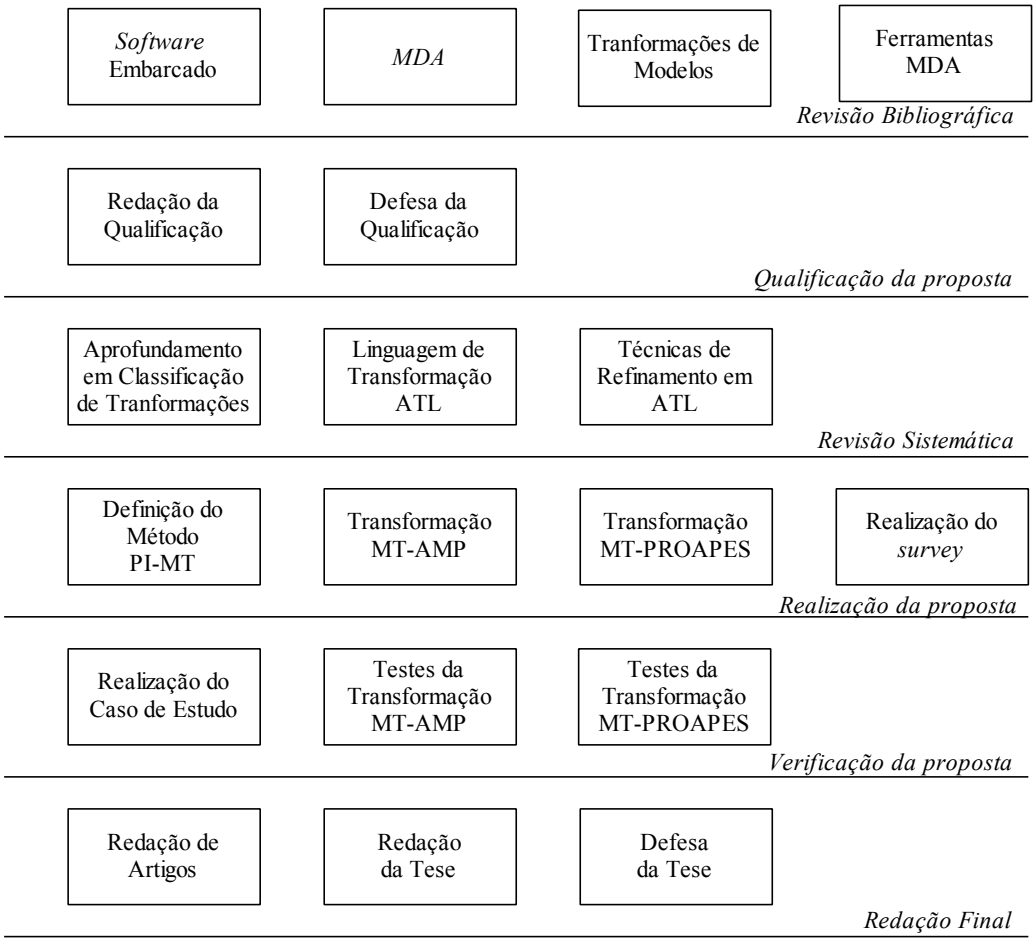

Figura 1 – Metodologia de Desenvolvimento

Um sistema Simulador de Alarme (SA) baseado em uma plataforma embarcada foi desenvolvido como caso de estudo. A verificação e testes das transformações MT-AMP e MT-PROAPES foi realizada utilizando-se a técnica de testes de "caixa preta" e os diagramas desenvolvidos para esse sistema foram utilizados como parte do conjunto de casos de testes. Em adição, diversos artigos foram escritos e submetidos para periódicos e eventos durante a realização desta pesquisa.

1.5 ORGANIZAÇÃO

O Capítulo 2 desta tese apresenta uma revisão bibliográfica sobre sistemas embarcados e Sistemas Operacionais em Tempo-Real. Esse capítulo também introduz a abordagem MDA, ressaltando os seguintes pontos: ciclo de desenvolvimento de *software* em MDA, vantagens e benefícios do uso da *MDA* e padrões relacionados com a MDA.

O Capítulo 3, por sua vez, trata de transformações de modelos no contexto da MDA. Também são abordados os seguintes temas relacionados com transformações de modelos: classificação das transformações de modelo, a linguagem de transformação de modelos ATL e o refinamento de modelos, um tipo específico de transformação e de especial interesse para esta pesquisa.

No Capítulo 4 é apresentado o método PI-MT, proposto nesta pesquisa, salientando os seguintes pontos: estrutura do método proposto, uso de Modelos de Plataforma explicitamente definidos como entrada da transformação e passos a serem executados para a realização do método. As transformações de modelos MT-AMP e MT-PROAPES implementadas, segundo as diretrizes do método PI-MT, também são apresentadas nesse capítulo. A MT-AMP tem como foco a modelagem estrutural do sistema de *software*, por sua vez, a MT-PROAPES é voltada para a modelagem comportamental. Finalizando o capítulo, são descritos a verificação e os testes realizados para as transformações implementadas.

Por fim, o Capítulo 5 descreve o caso de estudo realizado que contempla o desenvolvimento de um sistema embarcado de alarme voltado para plataformas baseadas em RTOS. As conclusões desta pesquisa são apresentadas no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os dois principais temas envolvidos com o desenvolvimento desta pesquisa: sistemas embarcados e *Model Driven Architecture* (MDA). Com relação ao tema de sistemas embarcados são abordados os seguintes conceitos: características e restrições de sistemas embarcados, *software* embarcado e Sistemas Operacionais em Tempo Real. Por sua vez, os conceitos relacionados com MDA apresentados neste capítulo são: ciclo de desenvolvimento de *software* em MDA, modelos e metamodelos e padrões OMG.

2.1 SISTEMAS EMBARCADOS

Sistemas embarcados são sistemas microprocessados de propósito específico. Estes sistemas realizam um conjunto de tarefas pré-definidas e, geralmente, são embarcados em um produto maior (MARWEDEL, 2006; STADZISZ e RENAUX, 2007). Desse modo, sistemas embarcados são sistemas especializados dedicados à execução de determinadas tarefas.

Exemplos de aplicações de sistemas embarcados são: controladores de tráfego aéreo, sistemas automotivos, equipamentos de robótica, sistemas aeroespaciais e eletrodomésticos em geral. A Figura 2 ilustra alguns desses exemplos.

Figura 2 – Exemplo de produtos embarcados

Atualmente, estima-se que existem aproximadamente 30 sistemas embarcados para cada habitante em países desenvolvidos, sendo que o mercado de sistemas embarcados cresce de 10 a 20% ao ano (EBERT e JONES, 2009).

Os sistemas embarcados são constituídos de diversos componentes e podem até ser implementados de forma integrada, em um único *chip* (SoC, acrônimo de *System-on a-Chip*), havendo funcionalidades implementadas tanto em *hardware* quanto em *software* (STADZISZ e RENAUX, 2007). Em sistemas embarcados, o *hardware* e o *software* estão fortemente relacionados e, desta forma, o *software* embarcado interage com o *hardware* que foi projetado especificamente para este sistema.

Sistemas embarcados são integrados em ambientes heterogêneos e interagem com o mundo real por meio de interfaces externas. Esta interação impõe requisitos de reatividade, exigindo respostas imediatas aos eventos capturados do mundo real. Desta forma, sistemas embarcados são frequentemente sistemas reativos, pois interagem continuamente com outros sistemas, produzindo reações a estímulos vindos do mundo exterior em um tempo pré-determinado (DOUGLASS, 1999; MARWEDEL, 2006).

Os principais requisitos de um sistema embarcado (propriamente ditos) são: recursos limitados, restrições de tempo-real, portabilidade, confiabilidade, escalabilidade, segurança e tolerância a falhas (STADZISZ e RENAUX, 2007; BERGER, 2002; DOUGLASS, 1999; MARWEDEL, 2006; WOLF, 2012). Além disso, sistemas embarcados podem estar sujeitos a diversas restrições como dimensão, peso e custo. É o caso de produtos portáteis que possuem restrições de dimensão e de peso, por exemplo, telefones celulares.

Vários produtos que utilizam sistemas embarcados são produzidos em larga escala, um exemplo disso são produtos eletrônicos de consumo. Assim, devido à grande concorrência existente neste mercado, a redução de custos torna-se fundamental, visando um menor preço final do produto. Isso se reflete principalmente no processo de criação do *hardware* a ser utilizado pelo sistema embarcado e resulta em uma restrição dos recursos a serem utilizados na sua construção (STADZISZ e RENAUX, 2007; MARWEDEL, 2006).

2.1.1 Arquitetura de Sistemas Embarcados

A arquitetura de um sistema embarcado é uma abstração do sistema e, desse modo, não representa informações detalhadas de implementação, como código-fonte ou projeto de *hardware*. Em um sistema embarcado, no nível de arquitetura, os componentes de *hardware* e

software são representados como um conjunto de elementos que interagem entre si. Esses elementos são representações de *hardware* ou *software* que consideram os aspectos de comportamento e de relacionamento entre esses elementos, abstraindo, assim, detalhes de implementação. Desse modo, uma arquitetura embarcada inclui elementos do sistema, elementos que interagem com o sistema, as propriedades de cada um dos elementos individuais e os relacionamentos entre estes elementos (NOERGAARD, 2005).

Um sistema embarcado deve considerar os seguintes aspectos arquiteturais: a arquitetura do sistema embarcado como um todo e as arquiteturas do *software*, do *hardware* e do processador (STADZISZ e RENAUX, 2007). Projetos de sistemas embarcados geralmente utilizam um modelo de arquitetura estruturado em três camadas: *hardware*, núcleo operacional (*kernel*) e *software* de aplicação, ilustrado na Figura 3 (NOERGAARD, 2005).

A camada de *hardware* contém os principais componentes do circuito eletrônico, enquanto as camadas de núcleo operacional e de *software* da aplicação contêm todo o *software* que será processado pelo sistema embarcado. Este modelo é uma representação modular de uma arquitetura de sistemas embarcados.

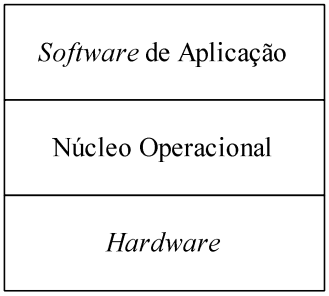

Figura 3 – Modelo de arquitetura de sistemas embarcados
Fonte: Adaptada de Noergaard (2005)

Entretanto, faz-se necessário refinar o modelo de três camadas, ilustrado na Figura 3, para representar sistemas embarcados complexos. Uma alternativa é utilizar o modelo de sete camadas, proposto por Góes e Renaux (2006), que define níveis de abstração do *hardware* agrupados da seguinte forma: *Hardware*, *Software Básico* e *Software de Aplicação*. As sete camadas definidas por esse modelo e descritas na Tabela 1 são: *Hardwired*-HW, *Softwired*-HW, *Low-Level Device Drivers*, Núcleo Operacional, Protocolos, Serviços e *Software* de Aplicação.

Tabela 1 – Arquitetura de sistemas embarcados em 7 camadas
Fonte: Góes e Renaux (2006)

Software de Aplicação	*Software* de Aplicação	Camada de *software* que implementa a funcionalidade específica de um determinado sistema embarcado.
Software Básico	*Low-Level Device Drivers*	Camada de *software* básico que interage com os dispositivos de *hardware* das camadas 1 e 2.
	Núcleo Operacional	Também denominado *kernel* ou Sistema Operacional em Tempo-Real (RTOS - *Real-Time Operating System*). Esta camada visa controlar o sistema de concorrência, interrupções e temporização.
	Protocolos	Camada de *software* responsável pelo controle de tráfego de informações de forma estruturada sobre os canais de comunicação disponíveis (*Ethernet*, USB, RS-232, RS-485, GPRS, entre outros).
	Serviços	Camada que implementa módulos de *software* de uso comum, como sistemas de arquivos, por exemplo.
Hardware	*Hardwired*-HW	Camada de dispositivos físicos na qual os sinais elétricos são comandados por dispositivos semicondutores para realizar uma determinada atividade.
	Softwired-HW	Camada de *software* armazenada em dispositivos lógicos programáveis. A programação destes dispositivos é tipicamente realizada em VHDL.

Os sistemas embarcados construídos com base no modelo de sete camadas, proposto por Góes e Renaux (2006), visam à utilização de módulos de *hardware* e *software* previamente desenvolvidos e testados, reduzindo, assim, o esforço de desenvolvimento pelas camadas 2 (*Softwired*-HW) e 7 (*Software* de Aplicação).

2.1.2 Arquitetura de Processadores

Arquiteturas de processadores podem ser diferenciadas em CISC (*Complex Instruction Set Computer*), RISC (*Reduced Instruction Set Computer*) e VLIW (*Very Long Instruction Word*) (STADZISZ e RENAUX, 2007). Processadores VLIW são utilizados, principalmente, em computadores de grande porte e, portanto, pouco utilizados em sistemas embarcados.

A arquitetura clássica dos processadores é representada pelos processadores CISC que definem operações complexas compostas de várias instruções. Exemplos de processadores que implementam arquiteturas CISC são: X86 da Intel e m68k da Motorola.

A arquitetura RISC define um conjunto menor e mais simples de operações desenvolvidas para poucas instruções. Esta arquitetura possui um número reduzido de ciclos por operação. Muitos processadores RISC possuem apenas operações de um ciclo, enquanto

arquiteturas CISC tipicamente possuem operações de múltiplos ciclos. ARM, PowerPC, SPARC e MIPS são exemplos de arquiteturas RISC (NOERGAARD, 2005).

A escolha da arquitetura dos processadores a serem usados na construção de sistemas embarcados deve considerar requisitos como desempenho e custo do produto. Atualmente, as principais arquiteturas de processadores utilizadas no desenvolvimento de projetos embarcados são: ARM, MIPS, Hitachi SH, PowerPC e x86 (JIE e GEN-BAO, 2010). Processadores ARM estão entre os mais comercializados atualmente e possuem 7 famílias de processadores (RISC): Cortex , ARM7 , ARM9 , ARM9E , ARM10E , ARM11 e SecurCore (HWANG *et al*., 2008).

Processadores embarcados suportam processamento de 4, 8, 16, 32 e 64 bits. Processadores de 128 bits suportam maior volume de dados e endereçam espaços maiores de memória. Entretanto, eles não são frequentemente usados em projetos de sistemas embarcados. Atualmente, os processadores de 32-bits são os mais utilizados em projetos de sistemas embarcados (JIE e GEN-BAO, 2010).

2.1.3 Desenvolvimento de Sistemas Embarcados

O projeto de sistemas embarcados, também denominado *hardware*/*software codesign*, necessita modelar a aplicação e a arquitetura de *hardware*. O objetivo é encontrar o produto com a combinação adequada de *hardware* e *software* que resulte em um produto mais eficiente de acordo com suas especificações.

O ciclo de desenvolvimento de um projeto de sistema embarcado, proposto por Berger (2002), identifica as seguintes fases de desenvolvimento:

- Especificação do produto: tem como objetivo identificar os requisitos funcionais e não funcionais do sistema, e deve englobar os aspectos multidisciplinares existentes nos sistemas embarcados e que não são encontrados em sistemas convencionais.
- Particionamento entre *hardware* e *software*: dado que um projeto embarcado envolve componentes de *hardware* e *software*, é preciso decidir qual parte da solução será realizada pelo *hardware* e qual parte será executada pelo *software*. Esta decisão é chamada de “particionamento” entre *hardware* e *software*.
- Iteração e implementação: nesta fase são realizadas as atividades de projeto simultâneo de *hardware* e *software*.

- Detalhamento do projeto de *hardware* e *software*: o projeto deve especificar os aspectos de *hardware* e *software* envolvidos com o sistema.
- Integração de *hardware* e *software*: o projeto deve definir a forma de integração dos componentes de *hardware* e *software*.
- Testes de aceitação: o projeto de sistemas embarcados envolve a realização de um conjunto de testes mais amplo que os testes realizados em um projeto de sistemas tradicional, visto que os testes para sistemas embarcados devem contemplar também o *hardware* envolvido na aplicação.
- Manutenção e atualização: depois da homologação do produto, é preciso prover a manutenção e atualização deste.

O desenvolvimento de sistemas embarcados é um processo complexo devido às características de heterogeneidade, restrições de projeto e requisitos de reatividade, inerentes a esse tipo de sistema. Além disso, um produto embarcado pode estar conectado a vários dispositivos externos, e cada dispositivo pode necessitar de protocolos de comunicação e rotinas de controle específicas (STADZISZ e RENAUX, 2007).

2.1.4 *Software* Embarcado

A grande variedade de plataformas de *hardware* disponíveis aumenta a complexidade no processo de desenvolvimento de *software*, dado que o *software* embarcado é desenvolvido especificamente para uma determinada plataforma. Desse modo, o *software* embarcado é um dos principais componentes de sistemas embarcados (LEE, 2002).

O *software* embarcado deve configurar a plataforma computacional para tratar requisitos físicos, tais como tempo de execução, consumo de energia e desempenho. A arquitetura da plataforma computacional engloba os aspectos referentes ao *hardware*, ao processador e ao sistema operacional. Isto contrasta com o conceito clássico de *software* que é a realização de funções como procedimentos, nas quais entradas são mapeadas em saídas. Para tal, a maioria dos projetos de sistemas embarcados utiliza um Sistema Operacional em Tempo-Real (RTOS - *Real-Time Operating System*) como suporte para a execução do *software* da aplicação.

Um sistema embarcado típico compreende algumas características como concorrência, processamento em tempo-real e recursos limitados. Estes requisitos são comumente apoiados por RTOS. Como resultado, o *software* embarcado é concebido e implementado para ser executado diretamente sobre o RTOS (BECKER *et al*., 2010).

2.1.5 Sistemas Operacionais em Tempo Real

Um sistema operacional de uso geral oferece mecanismos de execução concorrente de processos, que disputam entre si o uso de recursos computacionais. Por sua vez, um Sistema Operacional em Tempo Real oferece suporte aos sistemas embarcados por meio de mecanismos de comunicação, sincronização, temporização e exclusão mútua. As principais características que diferem um RTOS de um sistema operacional de uso geral são: escalabilidade, políticas de escalonamento e suporte para ambientes embarcados (DOUGLASS, 1999).

A escalabilidade do RTOS se dá pela sua estruturação, que utiliza uma arquitetura de *microkernel*. Nessa arquitetura, um módulo central (*microkernel*) é responsável pelo gerenciamento de memória, comunicação entre tarefas, escalonamento e serviços de temporização, enquanto as demais funcionalidades do núcleo operacional são implementadas em módulos de *software* externos ao *microkernel* (MARWEDEL, 2006).

O escalonamento pode ser definido como o mapeamento do conjunto de tarefas para intervalos de tempo de execução. O escalonamento em Sistemas Operacionais em Tempo-Real deve ser determinado por meio de uma política de prioridades que define qual tarefa será executada, quando mais de uma tarefa encontra-se pronta para a execução. O RTOS deve disponibilizar meios para configurar com mais detalhes o processo de escalonamento das tarefas (MARWEDEL, 2006).

As restrições temporais, em um Sistema Operacional em Tempo-Real, podem ser classificadas como: restrições temporais rígidas (*hard real-time*), restrições temporais fortes (*firm real-time*) e restrições temporais flexíveis (*soft real-time*). Restrições temporais rígidas exigem que as tarefas sejam realizadas no prazo determinado. Os dois outros tipos de restrição toleram atraso.

2.1.5.1 Exemplos de Sistemas Operacionais em Tempo Real

Dentre os RTOS utilizados no desenvolvimento de sistemas embarcados é possível citar: QNX Neutrino, VxWorks, Nucleus e X *Real-Time Kernel*; todos de código proprietário e amplamente utilizados em ambientes comerciais e acadêmicos. Outros exemplos de RTOS podem ser encontrados em (SALEM *et al*., 2008; AROCA e CAURIN, 2009).

O VxWorks (*Wind River* VxWorks) utiliza arquitetura de *microkernel* de tempo real. Desta forma, este RTOS pode garantir um tempo de resposta determinístico. O VxWorks é disponibilizado para as seguintes plataformas: ARM, MIPS, PowerPC, x86, dentre outros. O ambiente de desenvolvimento deste RTOS é baseado na plataforma Eclipse e é compatível com compiladores para as linguagens de programação C, C++, Java e Ada. O VxWorks é amplamente utilizado pela indústria aeroespacial, automotiva, sistemas de defesa, dentre outras (TAO e SONG, 2011; BEHNAM *et al*., 2008; AROCA e CAURIN, 2009).

O QNX Neutrino, por sua vez, tem como base a arquitetura de *microkernel* de tempo real com memória protegida. Este RTOS é compatível com padrão POSIX (*Portable Operating System Interface*) e oferece suporte para desenvolvimento nas linguagens de programação C, C++ e Java. O QNX Neutrino suporta a arquitetura ARM, MIPS, PowerPC, SH4, dentre outras (KRTEN, 2009). Utilizado em aplicações da área automotiva, médica e de automação, possui um recurso chamado *particionamento adaptativo* que permite a criação de restrições de processamento por meio de tarefas (AROCA e CAURIN, 2009).

Ainda, o Nucleus possui arquitetura de *microkernel* de tempo real, compatível com padrão POSIX e com as linguagens de programação C, C++ e *Assembler*. O Nucleus oferece suporte para os seguintes processadores: ARM, MIPS, PowerPC e SuperH, dentre outros (MENTOR GRAPHICS, 2012).

Por fim, outro exemplo de núcleo operacional é o RTOS chamado X *Real-Time Kernel*, de especial interesse para este trabalho de pesquisa e apresentado a seguir.

2.1.5.2 X Real-Time Kernel

O núcleo operacional *X Real-Time Kernel*, comercializado pela empresa brasileira eSysTech – *embedded Systems Technologies*, possui uma arquitetura de *microkernel* e está fundamentado na orientação a objetos e no princípio da adaptação do *kernel* à plataforma de *hardware* e à aplicação (ESYSTECH, 2012). Empregado em processadores PowerPC e ARM,

o RTOS X é voltado para o desenvolvimento de sistemas embarcados com severas restrições temporais e de recursos computacionais, denominados *Deeply Embedded Systems* (GÓES e RENAUX, 2006).

O X *Real-Time Kernel* é um núcleo em tempo real que oferece infraestrutura para gerenciamento de tarefas, sincronização e troca de mensagens em sistemas embarcados. Este RTOS é disponibilizado por meio de uma biblioteca de *software* que define o núcleo do X e oferece serviços compartilhados de gerenciamento de tarefas, temporização, sincronização e mensagens. Extensões do *kernel* podem ser utilizadas, agregando, assim, serviços à camada de aplicação. A camada de aplicação engloba as tarefas que implementam as funcionalidades específicas de cada sistema embarcado, como interações com usuários e algoritmos de controle.

Device drivers são módulos dedicados responsáveis pela interação do *kernel* com os recursos de *hardware*. Cada *device driver* implementa rotinas de acesso e controle de operação de determinado periférico (como um teclado ou um *display* LCD) ou de um recurso de *hardware* (como memória *flash*). Os *device drivers* oferecem APIs (*Application Program Interfaces*) para acesso às funcionalidades dos periféricos e recursos do *hardware*. Um módulo dedicado, dependente de *hardware* e denominado *BootLoader*, é responsável pela inicialização da execução do sistema (ESYSTECH, 2012).

O *kernel* do X disponibiliza seus serviços para a aplicação por meio de métodos de um objeto global denominado “os”. Para o gerenciamento de tarefas, o X *Real-Time Kernel* permite a programação de múltiplas tarefas concorrentes (*threads*). Todas as tarefas compartilham a mesma área de dados e *heap*.

A política de escalonamento utilizada pelo X *Real-Time Kernel* para distribuição do tempo do processador entre as tarefas é do tipo *Round Robin*. Assim, a execução das *threads* é realizada de forma sequencial a partir de sua ordem de criação.

Funções de envio, recepção e monitoração fornecem os serviços para manipulação de canais de comunicação no X *Real-Time Kernel*. O tratamento de interrupções de *hardware*, interrupções de *software* e o uso de semáforos também são mecanismos oferecidos por este RTOS. Outras funções de temporização possibilitam que as aplicações realizem ações em tempos determinados ou controlem suas ações em função do tempo.

O X *Real-Time Kernel* oferece suporte a TCP/IP[1], *flash filesystem*[2] e USB STACK[3]. O RTOS X também possibilita a integração de outros módulos opcionais, como novos *device drivers* e protocolos de comunicação.

2.2 ABORDAGENS DE DESENVOLVIMENTO DIRIGIDAS A MODELO

A Engenharia Dirigida a Modelos (MDE, acrônimo de *Model Driven Engineering*), também chamada de Desenvolvimento Dirigido a Modelos (MDD, acrônimo de *Model Driven Development*), surgiu da crescente complexidade do *software* (TEPPOLA *et al*., 2009). MDE é um termo utilizado para descrever "*abordagens de desenvolvimento de software em que modelos abstratos do sistema de software são criados e sistematicamente transformados em implementações concretas*" (FRANCE; RUMPE, 2007). Desse modo, em MDE todo artefato de *software* é considerado um modelo ou elemento de modelo.

Modelos auxiliam na compreensão de problemas complexos e em suas possíveis soluções por meio de abstração (SELIC, 2003). Portanto, sistemas de *software*, que muitas vezes são sistemas de engenharia complexos, podem se beneficiar do uso de modelos e de técnicas de modelagem.

A iniciativa MDE mais conhecida e adotada atualmente é a Arquitetura Dirigida a Modelos (MDA, acrônimo de *Model Driven Architecture*), proposta pelo OMG (*Object Management Group*) (FRANCE e RUMPE, 2007; AMELLER *et al*., 2010). O grupo OMG é constituído como uma organização internacional, sem fins lucrativos, que padroniza o uso de especificações da indústria de computação. A conformidade com estas especificações possibilita o desenvolvimento de aplicações heterogêneas para a maioria das plataformas de *hardware* e sistemas operacionais.

A principal diferença entre a MDA e a MDE é que a primeira concentra-se no uso de tecnologias padronizadas para o desenvolvimento de *software*, como a UML, enquanto a segunda possibilita o uso de linguagens de modelagem desenvolvidas para um domínio específico, não necessariamente seguindo padrões previamente determinados (BALASUBRAMANIAN *et al*., 2006). O uso de padrões assegura a conformidade desse *software* com o conjunto de tecnologias apoiadas pelo OMG, que, por sua vez, garante sua

[1] Conjunto de protocolos de comunicação para computadores ligados em rede.
[2] Sistema de arquivos projetado para armazenar arquivos em dispositivos de memória *flash*.
[3] Componente que possibilita a comunicação entre os dispositivos USB e seus respectivos *drivers*.

integração, quando necessário. Além disso, a MDA adota diferentes modelos a serem definidos durante o ciclo de desenvolvimento do sistema de *software* e diferencia esses modelos conforme o nível de abstração associado a cada modelo.

2.3 MODEL DRIVEN ARCHITECTURE (MDA)

A MDA é uma abordagem de desenvolvimento de *software* baseada na construção de modelos em alto nível de abstração e independentes de plataforma. Nessa abordagem, os modelos são os principais artefatos do processo de desenvolvimento de *software*, sendo utilizados no entendimento, na construção, na implantação, na operação e na manutenção do projeto de *software* (OMG, 2003).

A idéia chave da abordagem MDA é criar modelos do sistema de *software* em diferentes níveis de abstração. Modelos com maior nível de abstração estão mais distantes das particularidades de uma plataforma específica, enquanto modelos com menor nível de abstração estão mais próximos dos detalhes da plataforma. Os modelos iniciais, independentes de plataforma, são traduzidos em modelos intermediários, dependentes da plataforma adotada. A partir destes modelos, o código do sistema é gerado (SELIC, 2003).

Dentre os principais benefícios esperados com o uso da MDA estão o aumento da qualidade e da produtividade no processo de desenvolvimento de *software* (KLEPPE *et al*., 2003; SELIC, 2003; FRANCE e RUMPE, 2007). A melhoria da produtividade e da qualidade é de fundamental importância diante do aumento da demanda e da complexidade dos sistemas de *software* e da diversidade de plataformas existentes para sua implantação. Esses benefícios são obtidos por meio do incremento do valor dos artefatos de *software* (modelos), em termos da especificação das funcionalidades do sistema em alto nível de abstração e da redução do índice de obsolescência destes artefatos. Assim, quanto mais longo o tempo de uso de um modelo, maior será o retorno derivado do esforço inicial para a criação desse modelo. Isso reduz a sensibilidade dos modelos às mudanças que necessitam ser realizadas, sejam elas alterações nos requisitos do sistema ou migração do sistema de *software* para outras plataformas (ATKINSON e KÜHNE, 2003).

Na abordagem MDA, os engenheiros de *software* devem especificar as funcionalidades do sistema de *software*, mas não precisam definir os detalhes da plataforma de implementação, pois estas informações serão incorporadas durante o processo de transformação de modelos. Desse modo, a MDA permite ao desenvolvedor concentrar-se na

definição das funcionalidades e requisitos do sistema, sem se preocupar com detalhes de implementação e especificação da plataforma.

Ao permitir que os engenheiros de *software* possam projetar os sistemas em alto nível de abstração, a abordagem MDA pretende reduzir a complexidade do processo de desenvolvimento de *software*, abstraindo os detalhes da plataforma e, assim, melhorando a qualidade e a produtividade (KLEPPE *et al*., 2003). Segundo Mohagheghi e Dehlen (2008), a melhoria da qualidade de *software* é percebida principalmente na redução do número de falhas no *software*.

O incremento na produtividade, por sua vez, é obtido por meio do uso de modelos de alto nível de abstração como os principais artefatos no processo de desenvolvimento de *software*. Desse modo, o desenvolvedor não precisa especificar os detalhes da plataforma e nem as atividades propensas a erros e que podem conduzir a projetos com nível de qualidade inferior.

O *survey* intitulado “Estudo da utilização de modelagem no desenvolvimento de software para sistemas microprocessados (*software* embarcado)”, realizado durante o desenvolvimento desta tese, identificou o impacto do uso da MDA em diversas atividades de desenvolvimento de *software* embarcado com relação aos benefícios obtidos em termos de produtividade. No segmento de sistemas embarcados, os resultados apontam que o incremento da produtividade, obtido pelo uso da MDA, ocorre, principalmente pela melhoria da comunicação entre os componentes da equipe de desenvolvimento, melhor compreensão do problema e maior facilidade de documentação do projeto de *software* (AGNERc *et al*., 2012). Ainda, é pertinente ressaltar que resultados semelhantes foram obtidos por pesquisas similares, realizadas no segmento de desenvolvimento de sistemas de *software* de propósito geral, não considerando sistemas embarcados em específico (NUGROHO e CHAUDRON, 2008; MOHAGHEGHI e DEHLEN, 2008).

2.3.1 Modelos e Metamodelos

O modelo de um sistema de *software* pode ser definido como a descrição ou especificação desse sistema e de seu ambiente, com um propósito determinado (OMG, 2003). Kopetz (2008) define um modelo como uma simplificação da realidade com o objetivo de definir uma propriedade específica relevante para um determinado fim. No desenvolvimento

de sistemas de *software*, os modelos possibilitam um melhor entendimento e representação do problema e da solução (SELIC, 2003).

Na abordagem MDA um modelo é usado para a geração de outro modelo, sendo que esses modelos possuem diferentes níveis de abstração. Desse modo, o desenvolvimento de *software* segundo a abordagem MDA contempla a criação de modelos com diferentes níveis de abstração, bem como a transformação desses modelos, de acordo com as seguintes etapas de desenvolvimento (OMG, 2003):

1. Especificação de um Modelo Independente de Plataforma (PIM, acrônimo de *Platform Independent Model*);
2. Especificação de um Modelo de Plataforma (PM, acrônimo de *Platform Model*);
3. Seleção de uma plataforma específica para o sistema;
4. Transformação de um PIM em um Modelo Específico de Plataforma (PSM, acrônimo de *Platform Specific Model*), com base em uma plataforma específica;
5. Transformação do PSM em código do sistema de *software*.

A Figura 4 ilustra os modelos definidos e as transformações realizadas sobre esses modelos, durante o ciclo de desenvolvimento de *software* no contexto da MDA. Inicialmente, o Modelo Independente de Plataforma (PIM) define as funcionalidades do sistema de *software* em alto nível de abstração e não especifica detalhes da plataforma de implementação adotada. Subsequentemente, o modelo PIM deve ser transformado em um Modelo Específico de Plataforma (PSM). O código-fonte é o resultado da transformação do PSM com base em uma linguagem de programação específica.

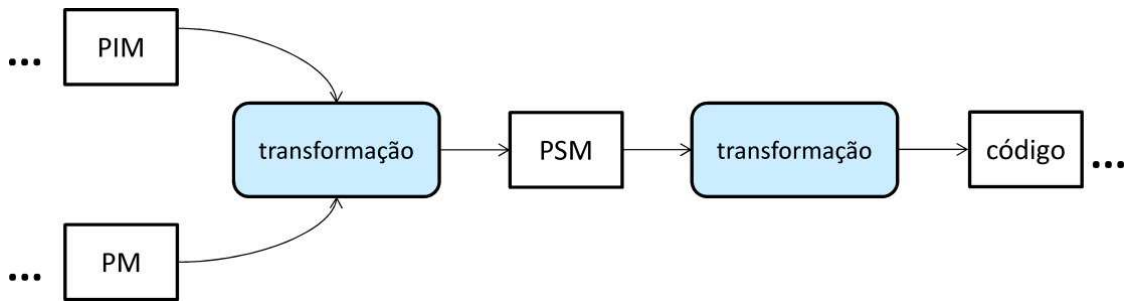

Figura 4 - Transformações MDA

2.3.1.1 Modelo Independente de Plataforma

O Modelo Independente de Plataforma (*Platform Independent Model* - PIM) é um modelo com alto nível de abstração e possui um nível suficiente de independência de plataforma. Por meio do PIM são definidas as funcionalidades do sistema de *software* (SIEGEL e OMG *Staff Strategy Group*, 2001).

Para atingir os benefícios da MDA, o modelo PIM deve suportar mudanças de tecnologia de seu ambiente de execução e possibilitar seu reuso em diferentes plataformas. Por consequência, o PIM é um artefato de fundamental importância no processo de desenvolvimento de *software*. O modelo PIM deve ser completo, no sentido que deve englobar a especificação do sistema de *software* por meio de diversos diagramas UML, como: diagrama de classes, diagrama de sequência e diagrama de estados.

Independência de plataforma é a capacidade de abstração de um modelo com relação às características das plataformas de execução. O guia da MDA (OMG, 2003) define plataforma como "*um conjunto de tecnologias que provê determinadas funcionalidades por meio de interfaces e padrões especificados, na qual um subsistema dependente da plataforma pode utilizá-la, sem preocupar-se com detalhes de como as funcionalidades fornecidas pela plataforma foram implementadas*". Assim, uma plataforma pode ser considerada como um meio para a execução de aplicações de *software* (OMG, 2001; OMG, 2003).

Embora a MDA classifique os modelos com base no nível de independência de plataforma, a literatura não explora de forma específica o que pode estar contido nos modelos PIM e PSM (ALMEIDA, 2006). Segundo os princípios da MDA, o modelo PIM deve possuir total independência de plataforma, abstraindo, desse modo, as características da plataforma. Entretanto, no desenvolvimento de diversos tipos de *software*, como *software* embarcado, isto não é possível, dado que o PIM necessita indicar os serviços da plataforma que serão utilizados pelo sistema de *software* (SELIC, 2005).

Outrossim, inicialmente, o termo "plataforma" referia-se apenas ao *hardware* de computação responsável por executar o *software*. Posteriormente, esta definição foi generalizada para incluir plataformas baseadas em *software*, tais como sistemas operacionais. Segundo SELIC (2005), um dos principais problemas em MDA é supor que independência de plataforma significa que os efeitos das plataformas podem ser totalmente ignorados durante o projeto de *software*. Independência de plataforma significa que um determinado projeto de *software* pode ser portado de uma plataforma para outra sem a necessidade de realizar

modificações no projeto. Entretanto, isto não significa que um projeto possa ser construído desconsiderando totalmente as questões relativas à plataforma.

No projeto de sistemas embarcados, geralmente suportados por um RTOS, plataformas são tema de grande preocupação, uma vez que as características da plataforma adotada são empregadas diretamente nas aplicações e, fundamentalmente, determinam seu comportamento e seu desempenho. Em adição, na construção de *software* embarcado esse problema se agrava devido à grande diversidade de plataformas existentes. Por esta razão, muitas vezes, é preciso referenciar aspectos do RTOS juntamente com a modelagem da aplicação (PIM) para possibilitar a correta representação do comportamento de um *software* embarcado, além de definir as restrições do sistema no início do projeto.

Desse modo, neste ponto em que se discute a "modelagem independente de plataforma", é preciso esclarecer que no desenvolvimento de sistemas embarcados a influência das plataformas não pode ser ignorada durante a definição das funcionalidades do *software*. Os efeitos das plataformas sobre o *software*, representado por meio de serviços do RTOS, necessitam ser representados na modelagem do PIM e direcionam o refinamento para níveis dependentes de plataforma.

Uma forma comum de se criar independência entre um modelo (como o PIM) e uma infraestrutura variável (como as plataformas) é criar uma camada de abstração que unifique (ou generalize) as referências a estas infraestruturas. Assim, no modelo são feitas referências genéricas utilizando a nomenclatura (ou linguagem) da camada de abstração ao invés de referências a uma infraestrutura em particular. Posteriormente, no momento da implementação de tal modelo (i.e., na sua transformação), as referências generalistas deverão ser substituídas ou mapeadas em referências da infraestrutura alvo, que deve ser compatível com o modelo generalista. Assim, o modelo (PIM) torna-se genérico, ou seja independente, para todas as variações da infraestrutura (plataformas) representadas pela camada de abstração usada como referência.

2.3.1.2 Modelo Específico de Plataforma

Modelo Específico de Plataforma (PSM, acrônimo de *Platform Specific Model*) é um modelo que combina as funcionalidades do *software* definidas no modelo PIM com detalhes que especificam como estas funcionalidades serão executadas pelos serviços oferecidos por

uma determinada plataforma. O PSM é o resultado da transformação do modelo PIM, com base em um modelo de plataforma específico.

O modelo PSM é gerado para especificar o sistema de *software* em termos dos construtores de implementação disponíveis para determinada tecnologia (OMG, 2001). A implementação de um sistema de *software* pode envolver diversas tecnologias, tais como: banco de dados, camada de apresentação *web*, servidor de aplicações, sistemas operacionais em tempo-real, dentre outras. Desse modo, um modelo PIM pode gerar diversos modelos PSM, relacionados com as diferentes tecnologias adotadas (OMG, 2003).

O PSM deve possuir as características necessárias para a geração do código-fonte da aplicação. Os elementos definidos no PSM devem especificar todos os detalhes necessários para a implementação das funcionalidades do sistema por uma determinada plataforma. Desta forma, é possível gerar o código-fonte de uma aplicação a partir de um modelo PSM.

2.3.1.3 Modelo de Plataforma

O Modelo de Plataforma (PM, acrônimo de *Platform Model*) descreve um conjunto de conceitos técnicos, representando os elementos que constituem a plataforma e os serviços que ela provê. O PM também deve especificar as restrições de uso destes elementos. SELIC (2005) define uma plataforma como um conjunto de mecanismos de *hardware* e *software* que possibilitam a execução de aplicações de *software*.

Atualmente, a maioria das abordagens de transformação de modelos define as características das plataformas implicitamente por meio das regras de transformação usadas para produzir o PSM a partir do PIM. Assim, essas abordagens não consideram um Modelo de Plataforma explicitamente definido e utilizado como entrada do processo de transformação. Consequentemente, deve existir um modelo de transformação configurado para cada plataforma que se pretende utilizar (CZARNECKI e HELSEN, 2003).

O uso de um Modelo de Plataforma explicitamente definido possibilita o reuso do processo de transformação de modelos e facilita a adoção de novas plataformas. Assim, obtêm-se a independência entre o processo de transformação de modelos e a plataforma de implementação utilizada (AGNERa *et al*., 2012; SOARESa *et al*., 2012).

2.3.1.4 Arquitetura de Metamodelagem

Um metamodelo é utilizado para definir uma linguagem de modelagem que descreve os tipos de elementos que podem ser utilizados para construir artefatos (modelos) nessa linguagem. A metamodelagem é fundamental na estrutura da abordagem MDA, pois oferece um mecanismo para definir linguagens de modelagem sem gerar ambiguidade (KLEPPE *et al*., 2003).

Metamodelos definem a estrutura sintática, a semântica e as restrições para um conjunto de modelos que compartilham sintaxe e linguagem semântica comuns. Um metamodelo é uma definição precisa dos construtores utilizados na criação de modelos. Desse modo, modelos são instanciados a partir de um metamodelo (MELLOR *et al*., 2005).

A Tabela 2 apresenta uma visão geral da arquitetura de metamodelagem que define quatro camadas de modelagem (OMG, 2003), sendo elas:

- Camada **M0**: representa as instâncias de modelos, ou seja, representação de objetos do mundo real ou objetos de controle e interface.
- Camada **M1**: representa os modelos usados para representar os sistemas e as aplicações.
- Camada **M2**: representa os metamodelos definidos a partir do meta-metamodelo MOF (*Meta-Object Facility*) e usados para definir modelos.
- Camada **M3**: representa o meta-metamodelo MOF. O MOF possui um conjunto de elementos usados para especificar os metamodelos e será apresentado na subseção 2.3.2.

Tabela 2 – Camadas da arquitetura de metamodelagem

Nível	Camada	Descrição
M3	Meta-metamodelo	Linguagem para definição de metamodelos
M2	Metamodelo	Linguagem para definição de modelos
M1	Modelo	Linguagem para definição de domínios de informação
M0	Objetos do usuário	Informações específicas de domínio

A Figura 5 ilustra as quatro camadas da arquitetura de metamodelagem definida pelo OMG (2003).

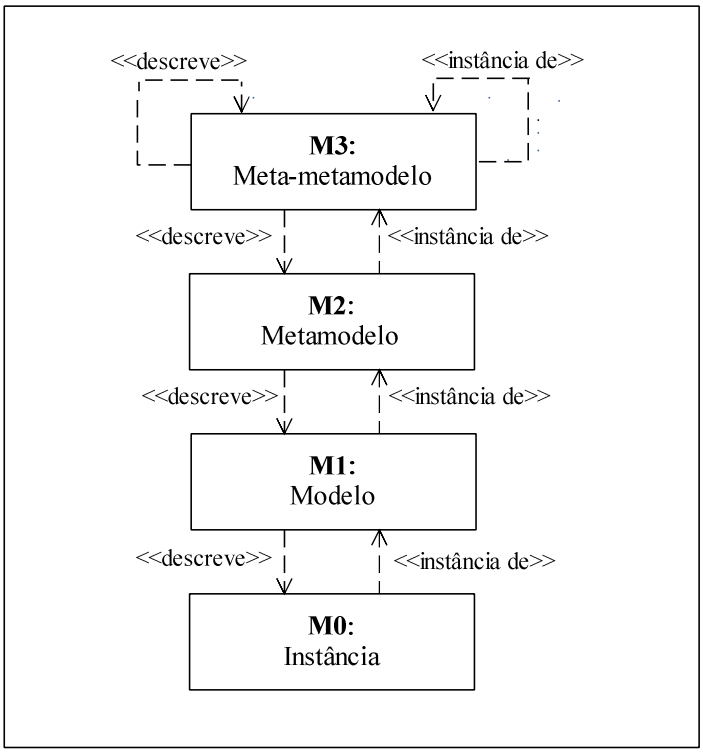

Figura 5 – Arquitetura de metamodelagem

2.3.2 Padrões OMG

A abordagem MDA está fundamentada no uso de padrões relacionados com a modelagem de *software*. O núcleo da MDA é baseado nas seguintes linguagens de modelagem adotados pelo OMG: MOF, UML, XMI e OCL. Os padrões MDA formam a base para a construção de esquemas coerentes para a criação, publicação e gerenciamento de modelos dentro de uma arquitetura dirigida a modelos.

2.3.2.1 Meta-Object Facility

O *Meta-Object Facility* (MOF) surgiu como uma iniciativa do OMG para padronizar a definição e o uso de metamodelos (DUBE e DIXIT, 2012). O MOF provê um repositório de metamodelos utilizado para especificar e manipular modelos, oferecendo consistência na manipulação destes em todas as fases da MDA (OMG, 2003). Desse modo, o MOF é um arcabouço de gerenciamento de metadados (dados que descrevem dados) e um conjunto de

serviços que possibilitam o desenvolvimento e interoperabilidade de sistemas dirigidos a modelos e metadados.

O MOF é um modelo orientado a objetos e possui um conjunto de elementos de modelagem utilizados na construção de metamodelos. São exemplos destes elementos: classes, associações, pacotes, tipos de dados, constantes, entre outros. Desse modo, o MOF define os elementos essenciais, sintaxe e estrutura de metamodelos utilizados para construir modelos de sistemas de *software*.

Ainda, o MOF é o meta-metamodelo que serve de base para a definição de diversas outras linguagens, tais como a *Unified Modeling Language* (UML), o *Common Warehouse Metamodel* (CWM) e *XML Metadata Interchange* (XMI), que residem na camada M2 da arquitetura de metamodelagem.

2.3.2.2 UML

A *Unified Modeling Language* (UML) é a linguagem de modelagem padrão adotada pela OMG para especificar, visualizar, construir e documentar sistemas de *software* (OMG, 2011a). A UML é uma linguagem de modelagem de propósito geral que pode ser aplicada a diversos domínios de aplicação. Também, esta linguagem fornece suporte para modelagem de propriedades estruturais e comportamentais dos sistemas de *software*. A definição da sintaxe da UML segue a arquitetura de metamodelagem adotada pelo OMG (2003), sendo o metamodelo UML uma instância do MOF (*Meta-Object Facility*).

2.3.2.3 XML Metadata Interchange

O *XML Metadata Interchange* (XMI) é um formato de intercâmbio de metadados, definido em termos do MOF, e utilizado para representar modelos em *eXtensible Markup Language* (XML). Documentos XML têm o objetivo de prover, de maneira simples e independente de plataforma, a interoperabilidade através do uso de cadeias de textos dotadas de descrições. O XMI utiliza o *Document Type Definition* (DTD) para validar os documentos XML, de acordo com seus respectivos metamodelos. A DTD define a estrutura dos elementos que podem ser descritos nos documentos XML.

O XMI é um padrão suportado por várias ferramentas nas mais diversas áreas de desenvolvimento de *software*, como especificação e modelagem de sistemas. Este formato define uma estrutura de documento que considera a relação entre os dados e seus correspondentes metadados. O padrão XMI pode ser utilizado, por exemplo, como um formato para intercâmbio de modelos UML.

2.3.2.4 OCL

Object Constraint Language (OCL) é uma linguagem textual para especificar consultas, restrições e operações em modelos UML. Estas restrições são especificadas como invariantes, pré-condições e pós-condições. O principal objetivo da OCL é aumentar a expressividade dos artefatos gerados pela UML.

A OCL faz parte do padrão UML e é uma linguagem baseada em expressões que estão limitadas ao escopo do modelo ao qual fazem referência (OMG, 2001). As expressões em OCL não são ambíguas, estas expressões normalmente especificam condições invariantes que precisam ser asseguradas pelo sistema que está sendo modelado. Os principais propósitos da OCL são:

- Especificar invariâncias sobre classes e tipos no modelo de classe;
- Descrever pré-condições e pós-condições sobre operações e métodos;
- Especificar restrições sobre operações.

As restrições OCL são definidas em uma linguagem textual com referências a elementos de diagramas UML. Como exemplo, considere o fragmento de um diagrama de classes apresentado na Figura 6. Este diagrama possui uma classe denominada *Conta* e que possui, dentre outros, um atributo *numero* do tipo inteiro, que identifica o número da conta do cliente. Entretanto, sem restrições adicionais, um objeto *Conta* pode assumir valores negativos para o atributo *numero*. A restrição OCL refere-se à classe *Conta* e restringe o intervalo permitido de valores do atributo *numero* somente para valores inteiros positivos.

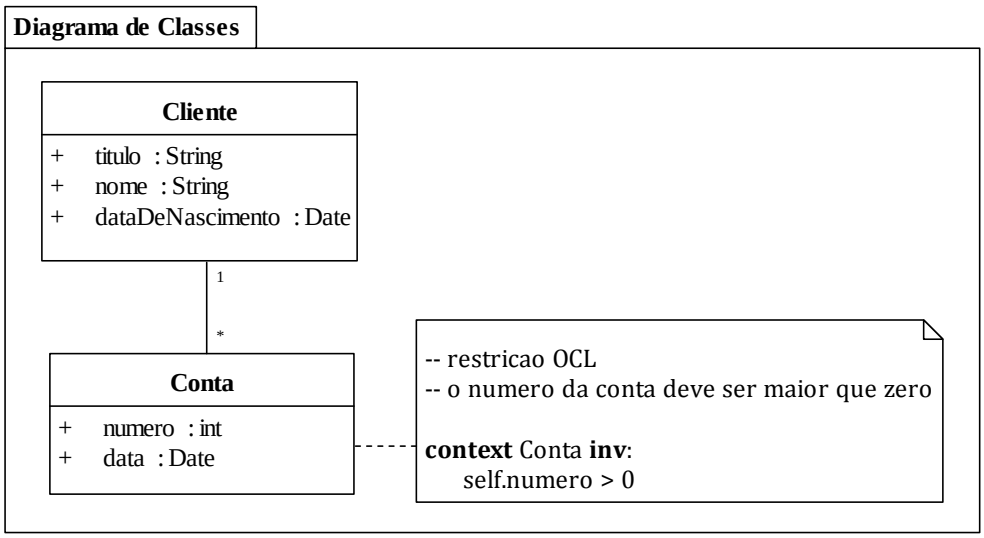

Figura 6 – Exemplo de restrição OCL

2.3.3 Perfil UML

A UML não oferece apoio para a modelagem de determinados sistemas de domínio específico. Entretanto, fornece uma base apropriada para criar perfis que definem plataformas de implementação e arquiteturas de *software* (ALTI *et al*., 2007). Um perfil permite personalizar a UML, especializando-a por meio de uma terminologia específica para um domínio de aplicação particular e adicionando novas semânticas e restrições.

Um perfil UML é definido por um conjunto de estereótipos especializados, valores rotulados (*tagged values*) e restrições. Estes elementos podem ser usados para adaptar a semântica da UML, sem alterar o metamodelo da UML (OMG, 2011a).

Estereótipo é um mecanismo de extensão da UML que define como uma metaclasse existente no metamodelo UML pode ser estendida. Um estereótipo habilita o uso de uma terminologia específica para um domínio ou plataforma em adição à terminologia usada para a metaclasse estendida (ERIKSSON *et al*., 2004).

Uma propriedade, também chamada de valor etiquetado (*tagged value*), é um meta-atributo vinculado a uma metaclasse do metamodelo UML estendida por meio de um perfil. Essas propriedades permitem adicionar informações aos elementos do modelo.

Restrições podem ser aplicadas a estereótipos, por exemplo, para definir as propriedades de boa formação de um modelo com o objetivo de assegurar que todas as relações entre os elementos são coerentes com o domínio da aplicação. Elas podem ser

expressas em qualquer linguagem (FUENTES-FERNÁNDEZ e VALLECILLO-MORENO, 2004), incluindo linguagem natural e OCL.

2.4 REVISÃO E CONCLUSÕES

Este capítulo apresentou os principais conceitos relacionados com sistemas embarcados, dentre eles: arquitetura de processadores, *software* embarcado e Sistemas Operacionais em Tempo Real. As restrições específicas que devem ser satisfeitas por sistemas embarcados, tais como eficiência no consumo de energia, confiabilidade, curto prazo para desenvolvimento e baixo custo, juntamente com a pressão incessante para aumentar as funcionalidades, leva a um significativo incremento da complexidade no desenvolvimento destes sistemas.

Um Sistema Operacional em Tempo Real (RTOS, acrônimo de *Real-Time Operating System*) é um sistema operacional empregado na construção de sistemas embarcados e possui características distintas, como: políticas de escalonamento, comunicação entre tarefas e serviços de interrupção. Dentre os RTOSs apresentados neste capítulo, destaca-se o X *Real-Time Kernel* que será utilizado como referência para a composição da família de plataformas usada como base das transformações de modelos implementadas por esta pesquisa. O X *Kernel* é um núcleo operacional em tempo real que oferece infra-estrutura para gerenciamento de tarefas, sincronização e troca de mensagens em sistemas embarcados.

A abordagem MDA é uma tendência e é apontada como uma solução potencial para o problema da complexidade no desenvolvimento dos sistemas de *software*, pois possibilita aos desenvolvedores atuarem em alto nível de abstração, preocupando-se somente com aspectos relacionados com as funcionalidades e os requisitos da aplicação (KLEPPE *et al*., 2003). Uma das principais etapas do processo de desenvolvimento de *software* no contexto da MDA é a transformação de modelos PIM, independentes de plataforma, em modelos PSM, específicos de plataforma.

O conceito de modelagem é fundamental para a abordagem MDA. A UML é a linguagem de modelagem padrão adotada pelo OMG para a modelagem de sistemas de *software*. Entretanto, existem situações nas quais a UML é muito genérica e possui um largo escopo, o que pode não ser apropriado para a modelagem de sistemas com domínio específico. A UML pode ser estendida para representar aspectos específicos de determinado domínio por meio dos seguintes mecanismos de extensão: estereótipos, valores rotulados e

restrições. Os novos elementos, definidos com base nestes mecanismos, compõem um perfil UML. A OCL, uma linguagem textual que permite definir restrições em modelos MDA, também foi apresentada.

Concluindo, o crescente aumento da complexidade no processo de desenvolvimento de sistemas embarcados beneficia-se do uso de abordagens dirigidas a modelos, como a MDA, por meio do uso de modelos de *software* de alto nível de abstração. A introdução de métodos, como o proposto por esta pesquisa, que efetivamente facilitem o uso de abordagens dirigidas a modelos pode melhorar significativamente a produtividade no processo de desenvolvimento de *software*, por meio da abstração dos aspectos referentes às plataformas e do consequente aumento da reusabilidade das transformações de modelos produzidas.

3 TRANSFORMAÇÃO DE MODELOS

Este capítulo trata do tema principal que envolve essa tese de doutorado: a transformação de modelos no contexto da MDA. Uma visão geral sobre a transformação de modelos é apresentada, seguida da classificação de transformações de modelos segundo a MDA.

Outro importante assunto tratado nesse capítulo é o refinamento de modelos, o principal tipo de transformação em abordagens dirigidas a modelo. A linguagem de transformação de modelos mais utilizada atualmente, a *Atlas Transformation Language* (ATL), também é apresentada. Por fim, esse capítulo introduz o ambiente TopCased, voltado para o desenvolvimento de *software* dirigido a modelos.

3.1 VISÃO GERAL SOBRE TRANSFORMAÇÃO DE MODELOS

A transformação de modelos é um tema fundamental em MDA e pode ser definida como a conversão de um modelo de mais alto nível de abstração para um modelo de mais baixo nível de abstração, com base em um conjunto de regras bem definidas (SINGH e SOOD, 2009). Segundo o OMG (2003), a transformação de modelos é o processo de convergir um modelo para outro modelo do mesmo sistema, com base em um mapeamento.

Por sua vez, um mapeamento é formado por regras de mapeamento, que descrevem uma correspondência entre os elementos do modelo-fonte com elementos do modelo-alvo. A transformação utiliza essa correspondência para, a partir de cada elemento do modelo-fonte, gerar os elementos correspondentes no modelo-alvo (SENDALL e KOZACZYNSKI, 2003; OMG, 2003).

Esta tese tem como foco a transformação de modelos PIM-para-PSM. O PIM e o PSM são artefatos do sistema e representam diferentes níveis de abstração na especificação da aplicação (KLEPPE *et al*., 2003). Por sua vez, neste âmbito, o mapeamento provê as especificações para a transformação de um PIM em um PSM com base em uma plataforma específica (OMG, 2003).

Em MDA, um mapeamento pode ser configurado por marcações, desta forma, diversos elementos do PIM podem ser marcados para indicar suas funções em determinado mapeamento. Este mapeamento é, então, utilizado para transformar os elementos PIM em

respectivos elementos do modelo PSM. O conceito de “marcação” de modelos será apresentado na subseção a seguir.

3.2 MARCAÇÃO DE MODELOS

A marcação de modelos é realizada por meio de “marcas” que permitem especificar informações não funcionais em um modelo-fonte. A marcação realizada no modelo PIM define o mapeamento a ser seguido para realizar a transformação de um modelo PIM em um modelo PSM. Desta forma, um elemento do modelo PIM é marcado para indicar qual mapeamento deve ser utilizado para transformá-lo em um ou mais elementos do PSM (OMG, 2003).

As marcas são específicas de uma determinada plataforma, portanto, não fazem parte do PIM original. O desenvolvedor de *software* marca o PIM para indicar dependências com relação a um metamodelo de plataforma (válido para uma família de plataformas). Desse modo, o PIM marcado é utilizado para criar um PSM específico para a tecnologia selecionada.

A inserção de marcas adiciona uma etapa ao processo de desenvolvimento de *software* dirigido a modelos. O modelo PIM é utilizado como entrada para o processo de transformação de modelos, entretanto, marcas devem ser adicionadas ao PIM antes da execução da transformação, criando um novo modelo.

Esse modelo, o PIM contendo marcas, é denominado PIM marcado (OMG, 2003). Finalmente, o PSM é gerado a partir do PIM marcado. A Figura 7 ilustra o processo de transformação que utiliza a marcação de modelos. Geralmente, a marcação de modelos é realizada por meio de marcas definidas em um perfil UML. Marcas providas por um perfil UML podem estar associadas a mais de um mapeamento (OMG, 2003).

Em MDA, as transformações de modelo seguem um padrão conhecido como padrão de transformação de modelos baseada em metamodelos, apresentado a seguir.

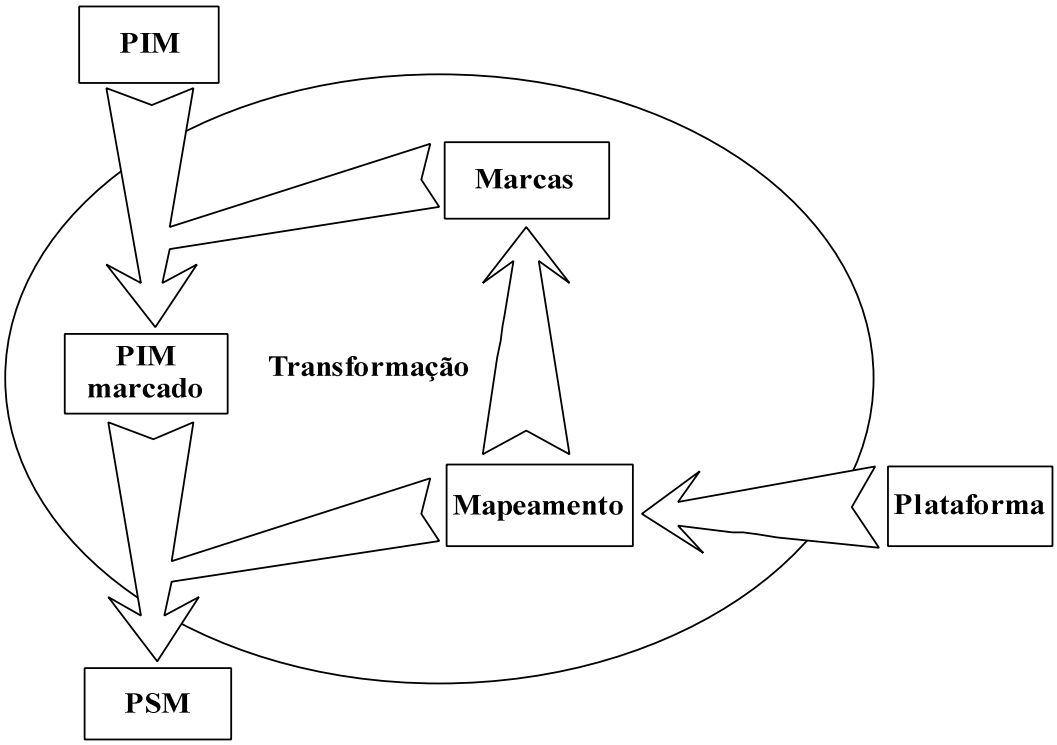

Figura 7 - O uso de marcas na transformação de modelos
Fonte: Adaptada de OMG (2003)

3.3 TRANSFORMAÇÕES BASEADAS EM METAMODELOS

Transformações de modelo segundo abordagens MDE/MDA seguem um padrão comum conhecido como “padrão de transformação de modelos baseada em metamodelos” (ALLILAIRE *et al*., 2006; BÉZIVIN *et al*., 2006). Conforme esse padrão, ilustrado na Figura 8, as transformações de modelo são definidas por meio de um modelo de transformação (BÜTTNER *et al*., 2011; BÉZIVIN *et al*., 2006). Desse modo, um modelo de transformação pode ser descrito como a relação entre os elementos do metamodelo-fonte e os elementos do metamodelo-alvo. Por sua vez, o modelo de transformação é definido com base em um metamodelo de uma Linguagem de Transformação de Modelos (MTL, acrônimo de *Model Transformation Language*).

A Figura 8 ilustra a transformação de modelos baseada em metamodelos. Os modelos fonte e alvo são criados com base em seus metamodelos, previamente definidos, e o mapeamento é executado entre esses metamodelos. Por sua vez, o modelo de transformação é definido com base no metamodelo de uma MTL e requer a configuração dos metamodelos fonte e alvo para a realização da transformação.

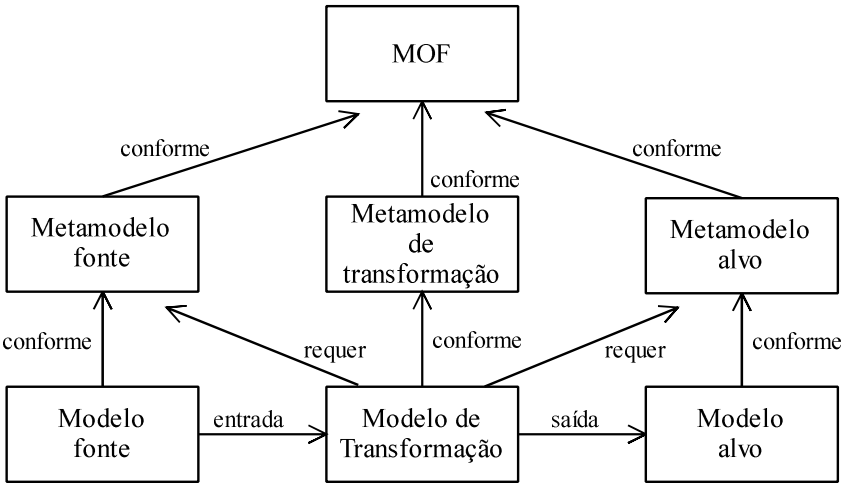

Figura 8 - Transformação de modelos baseada em metamodelos
Fonte - Adaptada de Bézivin (2006)

3.4 CLASSIFICAÇÃO DAS TRANSFORMAÇÕES DE MODELOS

As transformações de modelos são classificadas com relação ao nível de abstração dos modelos fonte e alvo, aos metamodelos (fonte e alvo) considerados e ao tipo dos artefatos produzidos pela transformação. Estas classificações serão apresentadas a seguir.

3.4.1 Transformações Horizontais e Verticais

Com relação ao nível de abstração, as transformações são classificadas em dois tipos distintos: horizontal e vertical (MENS e VAN GORP, 2006). Em transformações horizontais o modelo alvo, gerado pela transformação, é definido no mesmo nível de abstração do modelo-fonte. Segundo o OMG (2003), em MDA o termo "abstração" significa o processo de suprimir determinados detalhes de um modelo com o objetivo de se estabelecer um modelo mais simplificado. Por sua vez, transformações verticais referem-se a transformações nas quais o modelo-fonte e o modelo-alvo residem em diferentes níveis de abstração, sendo que o modelo-fonte é definido em um nível de abstração mais alto do que o do modelo-alvo.

Um exemplo típico de transformação vertical é o refinamento de modelos. Em um refinamento, um modelo abstrato é gradualmente refinado, por meio de transformações sucessivas que adicionam a este modelo detalhes mais concretos (EHRIG *et al.*, 2006; WIRTH, 1971). Por sua vez, transformações horizontais têm sido utilizadas para a migração

de modelos descritos em uma linguagem específica de domínio para outra linguagem, ambas definidas no mesmo nível de abstração (DANG e GOGOLLA, 2009).

3.4.2 Transformações Endógenas e Exógenas

Endógena e exógena são termos utilizados para classificar transformações com base no metamodelo utilizado como referência para expressar os modelos fonte e alvo. Transformações são endógenas quando o modelo-fonte e o modelo-alvo estão em conformidade com o mesmo metamodelo. Por sua vez, transformações exógenas são transformações realizadas entre modelos expressos utilizando-se diferentes metamodelos (MENS e VAN GORP, 2006).

O refinamento e a otimização de modelos são exemplos de transformações endógenas (SUN *et al*., 2009). Um exemplo de transformação exógena é a transformação de um modelo de *software*, descrito com base em uma linguagem de modelagem (baseada em um metamodelo de modelagem), em código do sistema, definido em uma linguagem de programação (baseada em um metamodelo distinto) (VAN GORP *et al*., 2007).

3.4.3 Transformações Modelo-Modelo e Modelo-Texto

Com relação aos artefatos produzidos, as transformações são classificadas como Modelo-Modelo (M2M) e Modelo-Texto (M2T) (CZARNECKI e HELSEN, 2006). Abordagens Modelo-Modelo transformam um modelo de entrada em um modelo de saída. A transformação de um PIM em um PSM é um exemplo típico de transformação M2M. Por sua vez, transformações Modelo-Texto produzem artefatos de texto a partir de modelos de entrada. Um exemplo de transformação M2T é a geração de código-fonte, que recebe um modelo de entrada (ex.: um diagrama de classes da UML) e realiza a geração dos arquivos de código-fonte descritos em determinada linguagem de programação (ex.: C++).

As ferramentas AndroMDA (TAENTZER *et al*., 2008) e Enterprise Architect (SPARK SYSTEMS, 2012) são exemplos de abordagens que implementam transformações Modelo-Texto. Por sua vez, dentre as abordagens M2M destacam-se as ferramentas baseadas em grafos e as linguagens de transformação de modelos (MTLs) baseadas em regras.

Transformações baseadas em grafos possibilitam uma manipulação visual de modelos de grafos com base em padrões e regras. São exemplos de abordagens de transformação baseadas em grafos: VIATRA (VARRÓ *et al*., 2002) e MOLA (KALNINS *et al*., 2004).

Ainda, MTLs baseadas em regras possibilitam a definição de regras declarativas e/ou imperativas para compor a transformação. Exemplos de linguagens baseadas em regras amplamente utilizadas são: ATL (JOUAULT *et al*., 2008), MT (TRATT, 2006), Epsilon (KOLOVOS *et al*., 2008) e QVT (VAN AMSTEL *et al*., 2011).

3.5 REFINAMENTO DE MODELOS

Abordagens dirigidas a modelos baseiam-se no refinamento passo a passo de modelos de análise em modelos de projeto cada vez mais detalhados, por todo o ciclo de desenvolvimento até a implementação (BRIAND *et al*., 2009). Um refinamento é uma transformação que adiciona detalhes específicos em um modelo existente (VAN DER STRAETEN *et al*., 2007).

Em um refinamento, grande parte do modelo-fonte é preservada, pois a maioria dos elementos do modelo-fonte é copiada para o modelo alvo, enquanto outros elementos devem ser alterados para incorporar características específicas. Desta forma, realizar um refinamento significa transformar um modelo abstrato em um modelo de projeto mais detalhado, ou seja, uma evolução *top-down* (BAUDRY *et al*., 2010).

Uma transformação de modelos também pode ser definida como um processo de geração de um modelo alvo a partir de um modelo-fonte, de acordo com uma definição de transformação expressa em uma linguagem de transformação de modelos (KURTEV, 2005). Linguagens de transformação de modelos especificam um vocabulário e uma gramática com semântica e regras bem definidas para a realização de transformações de modelos (DUBE; DIXIT, 2012).

Diversas MTLs foram propostas para definir e executar transformações no contexto da MDA, dentre estas se destaca a ATL (*Atlas Transformation Language*), a linguagem de transformação de modelos mais popular e largamente adotada atualmente (KALNINA *et al*., 2012; GOGOLLA e VALLECILLO, 2011; TOLOSA *et al*., 2011; TROYA e VALLECILLO, 2011).

3.6 ATLAS TRANSFORMATION LANGUAGE

A ATL (*Atlas Transformation Language*) é uma linguagem de transformação de modelos baseada em regras desenvolvida pelo INRIA-França (*Institut National de Recherche en Informatique et en Automatique*) em resposta a solicitação do OMG (*Object Management Group*) para proposta de uma linguagem de transformação de modelos compatível com o padrão QVT (*Queries / Views / Transformations*) (VAN AMSTEL *et al.*, 2011). Esta linguagem é amplamente reconhecida como padrão para o desenvolvimento de transformações de modelo no contexto da MDA (BÜTTNER *et al.*, 2011; TOLOSA *et al.*, 2011; TROYA e VALLECILLO, 2011; VARA *et al.*, 2009; ALLILAIRE *et al.*, 2006).

No contexto da ATL a definição dos modelos é realizada com base em seus metamodelos (JOUAULT e KURTEV, 2006), conforme ilustrado na Figura 9. Assim, as regras de transformação indicam claramente como os conceitos dos metamodelos de origem são mapeados nos conceitos do metamodelo alvo. Nesse padrão, um modelo-fonte é transformado em um modelo-alvo. A transformação é conduzida por uma definição de transformação (*modulo.atl*), definida com base nos construtores da linguagem ATL. Por sua vez, a definição da transformação também é um modelo. Os modelos fonte, alvo e a definição da transformação devem estar em conformidade com seus metamodelos (metamodelo-fonte, metamodelo-alvo e metamodelo ATL, respectivamente). Também, os metamodelos devem estar em conformidade com um meta-metamodelo, neste caso o MOF (OMG, 2011b).

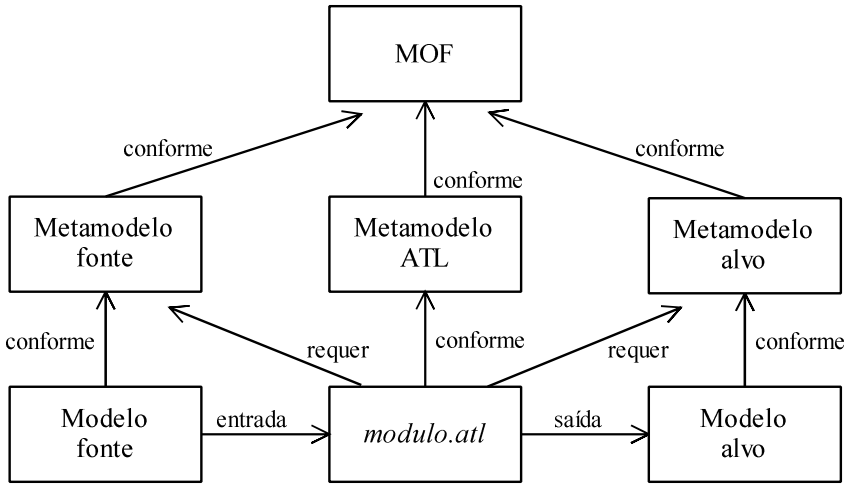

Figura 9 - Padrão de transformação de modelos da linguagem ATL
Fonte – Adaptada de Jouault e Kurtev (2006)

A ATL é uma linguagem híbrida, pois especifica construtores declarativos e imperativos. As construções declarativas são claras e precisas se comparadas com as imperativas e, portanto, mais usadas para escrever transformações. Estas construções permitem expressar correspondências entre os elementos do modelo-fonte e os elementos do modelo destino por meio de uma série de composições de regras. Adicionalmente, as construções imperativas possibilitam definir fluxos de controle e são usadas para especificar problemas complexos (FERNÁNDEZ-FERNÁNDEZ *et al*., 2008).

A ATL oferece um mecanismo de modularização que possibilita: 1) agrupar um conjunto de regras de transformação em módulos; e 2) realizar a combinação de módulos, por meio da importação de um módulo por outro módulo ATL para ter acesso a seu conteúdo (CZARNECKI e HELSEN, 2006). Desse modo, uma transformação ATL é formada por módulos estruturados com base nos seguintes elementos: *header* (cabeçalho), seção *import* (importadora), *helpers* e regras de transformação (TOLOSA *et al*., 2011). *Helpers* e regras são construtores usados para especificar as funcionalidades da transformação.

A seção *header* (obrigatória) define o nome do módulo de transformação e especifica os modelos fonte e destino. Por sua vez, estes modelos (fonte e alvo) devem estar relacionados com seus respectivos metamodelos. A Figura 10 mostra um exemplo de cabeçalho de um módulo de transformação denominado PIM2PSM.atl. O cabeçalho utiliza o modo de execução padrão, especificado por meio da palavra-chave “from”, e define como fonte (IN) o modelo PIM. O modelo de saída, denominado OUT, refere-se ao PSM, e é criado como resultado da transformação. Ambos os modelos baseiam-se no metamodelo UML2.

```
module PIM2PSM;
create OUT : UML2 from IN : UML2;
```

Figura 10 - Cabeçalho de um módulo ATL

A seção *import* permite importar bibliotecas ATL que definem um conjunto de funções de propósito geral, tais como funções de manipulação de *strings*. Um *helper* ATL é uma consulta definida com base na especificação OCL (*Object Constraint Language*) (OMG, 2010). *Helpers* realizam tarefas específicas e podem ser chamados em diferentes pontos de uma transformação ATL. Por sua vez, a OCL é uma linguagem usada para descrever expressões em modelos UML.

Uma regra de transformação pode ser considerada como uma função que implementa uma das etapas da transformação (CZARNECKI e HELSEN, 2006). Em ATL, as regras de transformação diferenciam-se entre: regras de combinação (*matched rules*) e regras de chamado (*called rules*). Regras de combinação seguem a abordagem declarativa e são executadas automaticamente. Uma regra de combinação especifica um mapeamento entre um conjunto de elementos do modelo-fonte e um conjunto de elementos do modelo alvo. Em oposição às regras de combinação, uma regra de chamado pode receber parâmetros e deve ser invocada a partir de um bloco imperativo ATL, a fim de ser executada. Desta forma, regras de chamado baseiam-se na abordagem imperativa e devem ser explicitamente invocadas por outra regra.

Existe um tipo específico de regra de combinação, denominada *lazy*. Uma regra *lazy* não é executada automaticamente como uma regra de combinação e, portanto, necessita ser explicitamente invocada por outra regra. Por sua vez, regras de chamado seguem a abordagem imperativa e, da mesma forma, devem ser explicitamente invocadas a partir de outra regra (CICCHETTI *et al*., 2007). A diferença entre regras *lazy* e regras de chamado é que as regras de chamado possuem uma especificação baseada em parâmetros, enquanto as regras *lazy* seguem uma especificação baseada em correspondência, como as regras de combinação. Uma correspondência estabelece uma relação entre elementos selecionados no modelo-fonte e elementos produzidos no modelo-alvo.

3.6.1 Ambiente de Execução de Transformações ATL

A ATL é compatível com o EMF (*Eclipse Modeling Framework*), um *framework* de modelagem para construção de ferramentas baseadas em modelos de dados estruturados (VAN AMSTEL *et al*., 2011). Para emprego da linguagem ATL, dispõe-se de um conjunto de ferramentas de desenvolvimento, denominado ADT (*ATL Development Tools*), composto do motor de transformação ATL e do ambiente de desenvolvimento integrado (IDE - *Integrated Development Environment*) que utiliza a API (*Application Programming Interface*) do Eclipse (ALLILAIRE *et al*., 2006). A arquitetura da linguagem ATL é ilustrada na Figura 11.

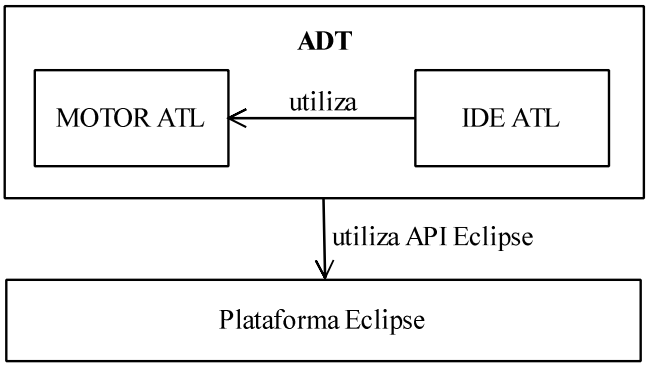

Figura 11 - Arquitetura ADT
Fonte - Adaptada de Allilaire *et al.* (2006)

A realização de uma transformação ATL envolve os seguintes passos: i) verificação de erros semânticos e sintáticos com relação ao metamodelo ATL e aos metamodelos de origem e de destino; ii) compilação; e iii) execução da transformação (JOUAULT e KURTEV, 2006). O motor de transformação ATL realiza tarefas como compilação e execução. O processo de compilação é disparado sempre que um arquivo ATL é modificado e salvo. Por sua vez, a execução de um módulo ATL é realizada em três fases, sendo elas:

- Fase de inicialização do módulo: primeiramente é realizada a inicialização dos atributos e, em seguida, são executadas as regras de chamado.
- Fase de correspondência de elementos do modelo-fonte: as condições de combinação são testadas com os elementos do modelo-fonte. Para cada combinação encontrada, o mecanismo ATL aloca o conjunto de elementos no modelo alvo.
- Fase de inicialização de elementos do modelo alvo: cada elemento de modelo alvo alocado é inicializado com base nas ligações associadas.

Em adição, o ambiente de desenvolvimento ATL oferece aos desenvolvedores recursos de depuração que possibilitam a execução da transformação de modelos passo a passo, inclusão de pontos de interrupção e exibição do conteúdo de variáveis. Além disso, a IDE ATL permite aos desenvolvedores visualizar as instruções ATL que estão sendo executadas durante a realização da transformação.

A ATL não impõe requisitos sobre a linguagem de modelagem usada, nem sobre o metamodelo utilizado como referência. Também, esta linguagem aceita diversos modelos como entrada do processo de transformação. Transformações ATL são unidirecionais e acessam o modelo-fonte no modo somente-leitura e o modelo-alvo no modo somente-escrita.

Em ATL é possível realizar um refinamento de modelos usando as seguintes abordagens: o modo *Refining* e a técnica de composição denominada *sobreposição de módulos*. Estas técnicas são detalhadas a seguir.

3.6.2 Modo *Refining*

O modo *refining* é um suporte explícito para realizar refinamentos em ATL na forma de um modo de execução (TROYA e VALLECILLO, 2011). Este modo é selecionado por meio do uso da palavra chave “refining” no cabeçalho (*header*) do módulo de transformação. O *refining* pode ser aplicado somente em transformações endógenas, quando os modelos fonte e destino utilizam o mesmo metamodelo. Neste modo de execução, a transformação cria elementos no modelo alvo do mesmo tipo dos elementos existentes no modelo-fonte. Então, todas as propriedades dos novos elementos são inicializadas com os mesmos valores definidos nas respectivas propriedades dos elementos fonte.

A Figura 12 apresenta o cabeçalho de uma transformação de refinamento que utiliza o metamodelo UML2 como metamodelo de referência para a definição dos modelos fonte e alvo. Neste modo de execução, a palavra-chave “refining” deve substituir a palavra-chave “from” no cabeçalho da transformação.

```
module PIM2PSM;
create OUT : UML2 refining IN : UML2;
```

Figura 12 - Modo *Refining*: Cabeçalho de Configuração

O compilador ATL2010 implementa a estratégia *in-place*, ou seja, as alterações são realizadas diretamente no modelo-fonte sem a necessidade de realizar a cópia dos elementos. Desta forma, as regras de transformação necessitam somente especificar as alterações a serem realizadas para gerar o novo modelo, enquanto os outros elementos permanecem inalterados. A versão do compilador ATL2010 oferece suporte à exclusão de elementos, o que não acontecia nas versões anteriores, ATL2006 e ATL2004.

A Figura 13 ilustra uma visão geral de transformações baseadas no modo ATL *refining* e em conformidade com o meta-metamodelo MOF. Nessa figura, o metamodelo da

UML é adotado para a definição dos modelos fonte e alvo. Ainda, a transformação de modelos (realizada no modo *refining*) é definida com base no metamodelo da ATL.

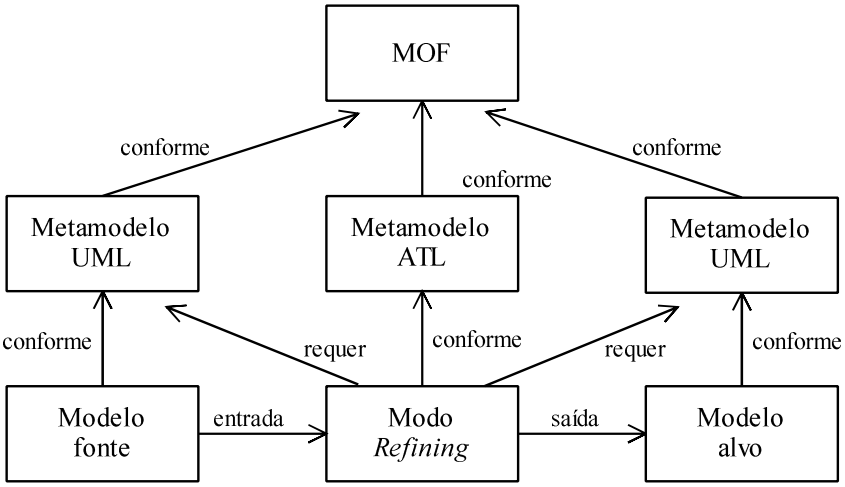

Figura 13 - Modo ATL *Refining*

Algumas características avançadas, oferecidas pelo modo de execução ATL padrão, não são suportadas pelo modo *refining*, dentre elas: uso de regras *lazy*, múltiplos elementos de entrada e blocos de ação (GRONMO *et al*., 2009; ECLIPSE, 2012a). Particularmente, um bloco da ação é uma sequência de sentenças imperativas que pode ser utilizado em regras de combinação (*matched*) e em regras de chamado (*called*).

As sentenças imperativas em ATL são os construtores usuais para definição de fluxos de controle, como condições e laços. Ainda, é pertinente ressaltar que no desenvolvimento de transformações mais complexas e que envolvam mais de um modelo de entrada, a falta destas características avançadas é bastante relevante e, muitas vezes, impossibilita o desenvolvimento de tais transformações.

3.6.3 Técnica de Sobreposição de Módulos

A Sobreposição de Módulos (*Module Superimposition*) é uma técnica de composição interna na qual um módulo de transformação é sobreposto por outro módulo de transformação (WAGELAAR, 2008). Desta forma, múltiplas definições de transformações são combinadas em uma única definição. Neste caso, as definições devem necessariamente usar a mesma linguagem de transformação de modelos.

A técnica de *sobreposição de módulos* permite que um módulo de transformação substitua determinadas regras do módulo de transformação sobreposto. Assim, a regra original é substituída por uma nova regra com o mesmo nome e contexto. Desta forma, esta técnica de composição permite dividir a transformação em módulos, o que resulta em melhoria da reusabilidade e da manutebilidade das transformações de modelos.

O módulo UML2Copy.atl, proposto por WAGELAAR *et al*. (2010), realiza a cópia de um modelo UML baseado no metamodelo UML2. Assim, a técnica de sobreposição pode usar este módulo na solução de problemas que envolvam o refinamento de modelos no contexto da MDA. Neste caso, as regras de transformação do módulo UML2Copy.atl podem ser reutilizadas em sua forma original em alguns casos e substituídas quando necessário por regras com o mesmo nome definidas no módulo específico de refinamento. O módulo UML2Copy.atl contém aproximadamente 200 regras, incluindo uma regra de transformação para cada metaclasse do metamodelo UML. De fato, as metaclasses do metamodelo UML são utilizadas para instanciar os elementos da UML (OMG, 2011a).

A Figura 14 demonstra um refinamento de modelos que utiliza a técnica de *sobreposição de módulos*. Nessa figura o módulo UML2Copy.atl contém as regras que realizam a cópia dos elementos do modelo-fonte para o modelo-alvo. Por sua vez, o *modulo.atl* é o módulo ATL que contém as regras específicas do refinamento que realizam alterações no modelo-fonte com base nos detalhes a serem incorporados no modelo-alvo.

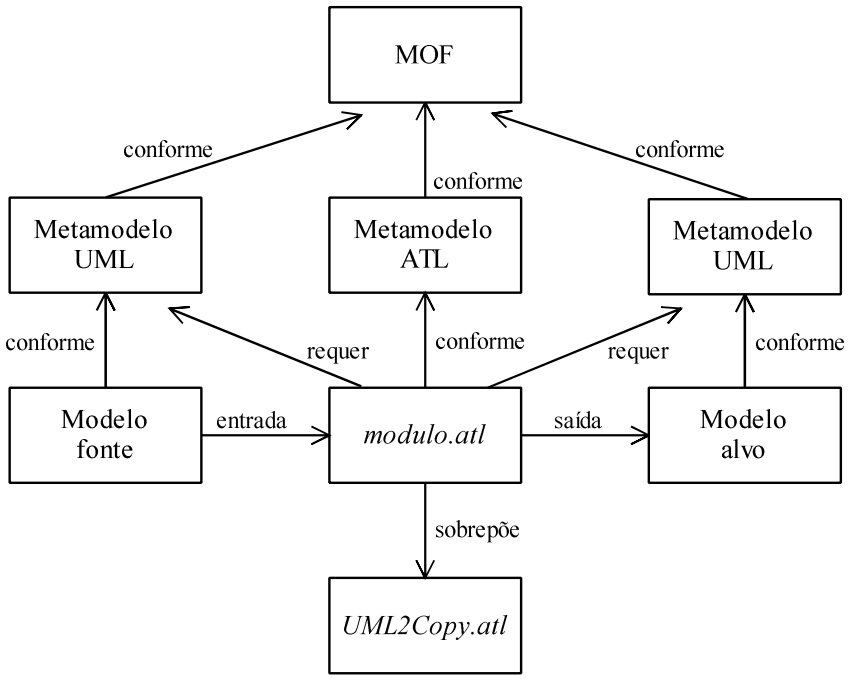

Figura 14 - Técnica de Sobreposição de Módulos

Os módulos de transformação devem estar em conformidade com o metamodelo de transformação. O metamodelo de transformação é a definição de uma linguagem de transformação de modelos (*Model Transformation Language* - MTL), neste caso a ATL (Figura 14).

3.6.4 Comparativo entra as técnicas *Refining* e Sobreposição de Módulos

Um comparativo entre as abordagens de refinamento ATL foi realizado durante o desenvolvimento desta pesquisa (AGNERb *et al*., 2012). Esse comparativo avaliou o modo *refining* e a técnica de *sobreposição de módulos* com relação às seguintes características:

1) **Execução *in-place***: modificações são realizadas diretamente no modelo-fonte, sem a necessidade de realizar a cópia dos elementos.
2) **Aplicação de perfil**: aplicar no modelo alvo os perfis relacionados com o modelo-fonte.
3) **Aplicação de estereótipos**: aplicar os estereótipos dos elementos do modelo-fonte em elementos do modelo-alvo.
4) **Melhor tempo de execução**: melhor desempenho na execução da transformação.
5) **Módulos de transformação menores**: número de linhas de código necessário para definir a transformação.
6) **Suporte a blocos de ação**: suporte ao uso de sentenças imperativas para definir características dos elementos alvo gerados.
7) **Suporte a regras *lazy***: suporte para regras *lazy*, regras invocadas por outras regras.
8) **Suporte a regras de chamado**: suporte para regras de chamado.
9) **Complexidade**: menor complexidade para o desenvolvimento e configuração da transformação de refinamento.
10) **Padrões alvo iterativos**: tornar possível gerar um conjunto de elementos do modelo alvo em conformidade com um mesmo tipo definido no modelo-fonte.

A Tabela 3 apresenta o suporte oferecido pelas abordagens avaliadas com relação às características analisadas.

Tabela 3 - Suporte oferecido pelas abordagens de refinamento ATL

Abordagens de refinamento ATL / Características / Suporte		*Refining* Mode	Sobreposição / UML2Copy
1	execução *in-place*	✓	
2	cópia de perfil	✓	
3	cópia de estereótipos	✓	
4	melhor tempo de execução	✓	
5	menor número de linhas de código	✓	
6	suporte a blocos de ação		✓
7	suporte a regras *lazy*		✓
8	suporte a regras de chamado		✓
9	complexidade	✓	
10	padrões alvo iterativos		✓

O modo *refining* realiza a cópia dos perfis aplicados ao modelo-fonte, bem como dos estereótipos aplicados aos elementos do modelo-fonte. Isto não ocorre com a técnica de *sobreposição de módulos* dado que, o módulo UML2Copy.atl não define regras no contexto de perfis e estereótipos. Desse modo, tais regras precisam ser definidas, quando necessário, o que acrescenta complexidade ao processo de desenvolvimento da transformação.

Conforme demonstrado por Tisi *et al*. (2011) o tempo de execução da transformação é menor no modo *refining* se comparado ao tempo obtido pela técnica de *sobreposição de módulos*. O tempo de execução do modo *refining* é menor, pois não é necessário realizar a cópia dos elementos do modelo-fonte que não serão modificados. Ainda, pertinente ressaltar que o tempo de execução da transformação é um aspecto relevante para se obter um desempenho adequado para ferramentas CASE (*Computer-Aided Software Engineering*), por exemplo. Também, o modo *refining* permite definir regras de transformação com menor número de linhas e comandos, pois não é necessário realizar a cópia das propriedades e referências que irão permanecer inalteradas (TISI *et al*., 2011).

Além das vantagens de melhor desempenho e menor tamanho do código da transformação, no modo *refining* a programação das transformações de refinamento é mais simples e fácil de usar, pois não requer domínio do módulo UML2Copy.atl, nem requer a configuração avançada da *sobreposição de módulos*. Por sua vez, o modo *refining* apresenta limitações que impedem, muitas vezes, o desenvolvimento de transformações mais complexas, como as que definem construtores imperativos. Também, atualmente o modo *refining* pode ser utilizado somente para transformar um único modelo-fonte em um único modelo-alvo (ECLIPSE, 2012b). Em oposição, a técnica de *sobreposição de módulos* não

possui estas restrições e pode tratar situações específicas, tal como utilizar múltiplos modelos de entrada (WAGELAAR *et al*., 2010).

3.6.4.1 Discussões

Os principais benefícios demonstrados pelo modo *refining* são: a simplicidade de uso e rapidez na execução. Entretanto, esta técnica possui severas restrições como: falta de suporte ao uso de blocos de ação e regras *lazy*. Estas restrições dificultam e muitas vezes impedem o desenvolvimento de transformações de modelos mais complexas e que usam mais de um modelo de entrada.

Os principais benefícios obtidos com o uso da técnica de *sobreposição de módulos* são a melhoria da manutenibilidade e da reusabilidade, obtidas por meio da composição de módulos e da superposição de regras em um mesmo contexto. Também, esta técnica mostrou-se mais flexível e eficiente, visto que não possui as limitações apresentadas pelo modo *refining*. Entretanto, é relevante ressaltar que a técnica de *sobreposição de módulos* requer o domínio do desenvolvedor com relação ao funcionamento da configuração de técnicas de composição em ATL e ao módulo UML2Copy.atl. Também, o módulo UML2Copy.atl não define regras para realizar a cópia da aplicação de um perfil a um modelo, tão pouco especifica regras que apliquem no modelo alvo os estereótipos existentes no modelo-fonte.

Desta forma, conclui-se que a escolha por uma das técnicas apresentadas deve ser avaliada caso a caso, dependendo das características e requisitos da transformação que se pretende desenvolver.

De acordo com o conhecimento da autora, as técnicas comparadas são as mais difundidas e adotadas na realização do refinamento de modelos em ATL. Outros trabalhos exploram a implementação do refinamento de modelos em outras linguagens de transformação de modelos (GUERRA *et al*., 2011; GOLDSCHMIDT e WACHSMUTH, 2008; KOLOVOS *et al*., 2008). Entretanto, estas pesquisas não são direcionadas para a linguagem ATL.

Ainda, a pesquisa realizada por TISI *et al*. (2011) abordou o refinamento de modelos usando linguagens baseadas em regras, como a ATL. Entretanto, foram considerados apenas os seguintes critérios: desempenho e tamanho final do código. Portanto, não foram explorados todos os critérios aqui avaliados.

3.7 TOPCASED

O ambiente TopCased pode ser utilizado para a construção e validação das transformações de modelos no contexto da abordagem MDA. O TopCased funciona como um interpretador para um subconjunto da UML e da OCL. Esse ambiente realiza a validação de modelos por meio da exploração, avaliando se um modelo cumpre os requisitos exigidos pela aplicação. Em adição, o TopCased realiza a validação de restrições OCL especificadas em um modelo. Desse modo, a OCL é utilizada pelo TopCased para verificar a consistência de modelos UML.

O TopCased é voltado para o desenvolvimento e execução de transformação de modelos que se destaca por integrar as principais tecnologias MDE/MDA, como a UML e a ATL, bem como por possibilitar a verificação de modelos UML e a validação das transformações de modelos realizadas.

O ambiente TopCased (***T****oolkit in* ***OP****en source for* ***C****ritical* ***A****pplications &* ***S****yst****E****ms* ***D****evelopment*) é composto por um conjunto de ferramentas de código aberto para a engenharia de sistemas, de *software* e de *hardware*, cujo principal objetivo é possibilitar o desenvolvimento de métodos e ferramentas integradas para o desenvolvimento de sistemas embarcados utilizando abordagens dirigidas à modelo (PONTISSO e CHEMOUIL, 2006).

O TopCased é baseado em tecnologias MDE, tanto para definição de modelos do sistema como para construção de um ambiente propício para a transformação de modelos em si. A validação e a verificação de modelos do sistema, bem como das transformações realizadas sobre estes modelos, são elementos chave no TopCased. Neste ambiente é possível verificar se uma transformação preserva as propriedades do modelo e, quando necessário, rastrear a execução da transformação para demonstrar que a propriedade não é garantida (CRÉGUT *et al*., 2010).

O projeto TopCased teve início em 2005, dedicado a sistemas da área automotiva, aeronáutica e de sistemas embarcados espaciais. Atualmente, o TopCased é um ambiente de código-fonte aberto, baseado em Eclipse, modular e que oferece um ambiente CASE (*Computer-Aided Software Engineering*) para o desenvolvimento de aplicações dirigidas a modelos, permitindo a modelagem do sistema, especificação da arquitetura do sistema e a implementação do *software* e do *hardware* (CRÉGUT *et al*., 2010).

O ambiente TopCased é organizado em torno de um modelo de arquitetura que permite a comunicação e o sincronismo entre a edição, a execução e a verificação de

transformação de modelos (BERTHOMIEU *et al.*, 2008). As tecnologias MDE utilizadas no TopCased são centradas em torno do EMF (*Eclipse Modeling Framework*).

Para apoiar o desenvolvimento dirigido a modelos, este ambiente inclui editores gráficos compatíveis com o metamodelo EMF, ferramentas de verificação e análise de modelos baseadas em OCL e linguagens de transformação de modelos, como a ATL. Também, metamodelos e editores para linguagens de modelagem mais utilizadas, tais como a UML e a SysML (*Systems Modeling Language*), são fornecidos pelo TopCased.

A Figura 15 apresenta a estrutura em camadas do TopCased, desenvolvido com base no Eclipse RCP (*Rich Client Platform*), EMF (*Eclipse Modeling Framework*) e GEF (*Graphical Editing Framework*). O EMF é um ambiente de modelagem e de geração de código para construção de ferramentas e aplicações baseadas em um modelo de dados estruturado. A abordagem de modelagem EMF é baseada no Ecore, um metamodelo definido com base no padrão MOF. Por sua vez, o GEF oferece suporte para a criação e uso de editores gráficos e visões no Eclipse.

O TopCased SDK (*Software Development Kit*), baseado no EMF Ecore, suporta o uso do metamodelo UML2 e de abordagens de transformação de modelos M2M (Modelo para Modelo), por meio de linguagens de transformação de modelos como ATL e QVT. Por fim, o TopCased oferece diversas ferramentas que dão suporte ao desenvolvimento de sistemas como editor UML, verificadores de regras, geradores de código, máquinas de rastreabilidade.

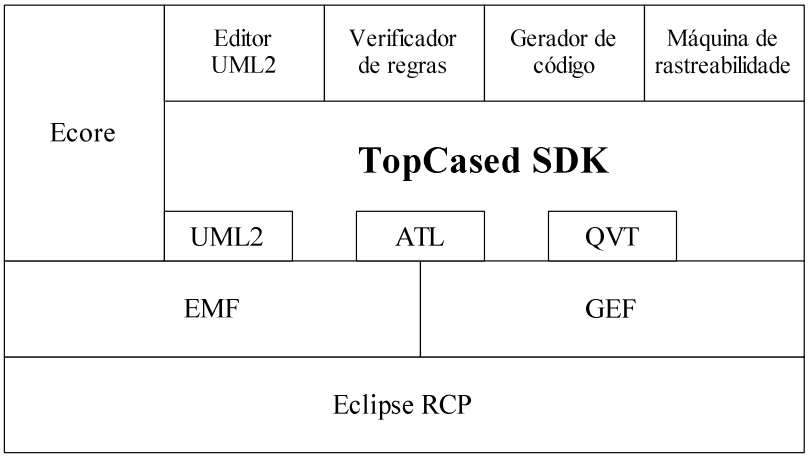

Figura 15 - Estrutura do ambiente TopCased
Fonte – Adaptada de Farail (2012)

3.8 CONCLUSÕES DO CAPÍTULO

O ciclo de desenvolvimento de *software* em MDA envolve transformações de modelos PIM para PSM e de PSM para código. Segundo KLEPPE *et al*. (2003), a etapa mais complexa no processo de desenvolvimento MDA é a transformação de um modelo PIM em um modelo PSM. Este processo define as especificações para transformação de um PIM em um PSM relacionado com uma plataforma específica.

Este capítulo apresentou os principais conceitos envolvidos com a transformação de modelos no contexto da MDA, dentre eles: marcação de modelos, classificação das abordagens de transformação de modelos, refinamento de modelos e a linguagem ATL. A ATL é uma linguagem amplamente utilizada para resolver problemas de transformação de modelos em MDA.

Esta pesquisa avaliou a técnica de composição denominada *sobreposição de módulos*, cujos principais benefícios são a melhoria da manutenibilidade e da reusabilidade, obtidas por meio da composição de módulos e da sobreposição de regras em um mesmo contexto. Esta técnica mostrou-se mais flexível e expressiva (blocos de ações), visto que não possui as limitações apresentadas pelo modo *refining*. Entretanto, vale ressaltar que a técnica de *sobreposição de módulos* é mais complexa de ser implementada, bem como, requer um tempo maior de processamento.

Por sua vez, os principais benefícios demonstrados pelo modo *refining*, outra técnica de refinamento avaliada nesta tese, são: a simplicidade de uso e rapidez na execução. Entretanto, o modo *refining* não suporta o uso de blocos de ação e de regras *lazy*. Estas restrições dificultam e muitas vezes impedem o desenvolvimento de transformações de modelos mais complexas e que utilizam mais de um modelo de entrada. Desta forma, conclui-se que a escolha por uma das técnicas apresentadas deve ser avaliada caso a caso, dependendo das características da transformação que se pretende desenvolver.

Este capítulo também apresentou o ambiente TopCased voltado para o desenvolvimento de transformações de modelos que envolvem sistemas embarcados críticos. Este ambiente foi escolhido para realizar os experimentos desta tese, visto que ele integra as principais tecnologias MDA adotadas pelo OMG, dentre elas a UML e a ATL.

4 MÉTODO PI-MT

Este capítulo descreve, como tema principal, o método *Platform Independent - Model Transformations* (PI-MT) proposto nesta tese. Os objetivos, os elementos e os passos de aplicação do método são detalhados. Em seguida, duas transformações de modelos construídas segundo as diretrizes do método proposto são apresentadas: MT-AMP e MT-PROAPES. A primeira suporta o uso de diagramas estruturais do sistema de *software*, por sua vez, a segunda é voltada para os diagramas comportamentais. Por fim, os testes realizados sobre essas implementações são descritos.

4.1 VISÃO GERAL DO PI-MT

Esta pesquisa propõe um método para criação de transformações de modelos no contexto da MDA, denominado *Platform Independent - Model Transformations* (PI-MT). Este método visa oferecer um mecanismo para criação de transformações reutilizáveis e aplicáveis a diferentes plataformas baseadas em RTOS. O PI-MT provê independência entre as transformações PIM-para-PSM e as características de plataforma, por meio do uso de um Modelo de Plataforma (PM) explicitamente definido (AGNERa *et al*., 2012; AGNER *et al*., 2011). De fato, conforme salientado anteriormente, o desenvolvimento de transformações utilizando PMs como modelos de entrada, permite a criação de transformações independentes de plataforma, sendo um importante tópico de pesquisa a ser explorado em MDA (SELIC, 2005).

A grande maioria das pesquisas em MDA, entretanto, preocupa-se em prover a "independência de plataforma" na etapa de modelagem da aplicação, desse modo, apenas os modelos do *software* (produzidos nas etapas iniciais do ciclo de desenvolvimento) são ditos "independentes de plataforma". Nesses casos, as transformações são dedicadas, i.e., incorporam as características da plataforma juntamente com as regras de transformação. Dessa forma, o problema da independência de plataforma é deslocado da etapa de modelagem para a etapa da transformação (TRATT, 2005; WAGELAAR e JONCKERS, 2005).

Este trabalho de pesquisa destaca-se por oferecer "independência de plataforma" tanto no nível de modelagem da aplicação quanto no nível da transformação de modelos. As próximas subseções tratam da independência de plataforma em *software* embarcado e da sua

relação com o desenvolvimento de transformações de modelos, fornecendo, assim, o embasamento necessário para detalhar o método proposto nesta tese.

4.2 INDEPENDÊNCIA DE PLATAFORMA

O termo “plataforma” se refere a um conjunto de mecanismos de *hardware* e *software* que suportam a execução de aplicações de *software*. Neste contexto, “permitir a execução” significa prover mecanismos externos ou serviços utilizados pelos sistemas de *software* (SELIC, 2005).

Os serviços de uma plataforma são tipicamente acessados por meio de uma Interface de Programação de Aplicativos (*Application Programming Interface* - API). Uma API é uma interface definida por um *software* para possibilitar o uso de suas funcionalidades por programas aplicativos (THOMAS *et al*., 2007). Por consequência, uma plataforma pode também ser visualizada como um mecanismo de abstração. No entanto, o objetivo principal de uma plataforma não é abstração, mas sim, permitir o acesso a suas funções e serviços. A abstração é, neste caso, apenas uma forma conveniente de habilitar o uso dos serviços oferecidos pela plataforma.

É importante compreender que uma plataforma é diferente das aplicações que ela suporta. Embora os serviços da plataforma sejam utilizados para a realização de uma aplicação, estes serviços não são parte da aplicação. Em plataformas compostas por sistemas operacionais, por exemplo, apesar da aplicação e o sistema operacional que a suporta estarem fortemente relacionados durante a execução, a aplicação não é parte do sistema operacional, nem o sistema operacional é parte da aplicação. Desse modo, é possível afirmar que as plataformas são independentes das aplicações que as utilizam, i.e., sua disponibilidade e operação não dependem da existência ou funcionamento das aplicações.

Desse modo, independência de plataforma significa que um determinado projeto de *software* pode ser portado de uma plataforma para outra sem a necessidade de realizar modificações no projeto. Isto não significa, entretanto, que um projeto possa ser construído desconsiderando totalmente as questões relativas à plataforma. Por exemplo, no âmbito de sistemas embarcados, muitas vezes é preciso definir aspectos do RTOS juntamente com a modelagem do sistema para possibilitar a representação do comportamento em tempo real de um *software* embarcado, bem como, definir as restrições do sistema no início do projeto.

4.3 INDEPENDÊNCIA DE PLATAFORMA E *SOFTWARE* EMBARCADO

Para se obter “independência de plataforma” em *software* embarcado é preciso definir a plataforma que se pretende utilizar de forma abstrata, considerando seus atributos de interesse. Por consequência, um projeto desenvolvido com base em uma plataforma abstrata pode ser portado de forma transparente para uma plataforma concreta, desde que satisfaça as restrições definidas na plataforma abstrata. Isto significa que um projeto de *software* deve realizar a modelagem da aplicação com base em um modelo de plataforma abstrato e, também, deve definir as ligações entre a modelagem da aplicação e a plataforma selecionada (SELIC, 2005).

Na área de sistemas embarcados, a diversidade de ambientes de execução (plataformas) é tão grande que o desenvolvimento de modelos universais, ou seja, totalmente independentes de plataforma, é questionável. Na percepção da autora desta pesquisa, as diferenças entre plataformas é tal que não é viável criar camadas de abstração que abstraiam ou generalizem todas as combinações de tecnologias de *hardware* e *software* existentes. Mesmo que se optasse por um subconjunto menor de plataformas envolvendo apenas alguns RTOS e tecnologias eletrônicas, uma camada de abstração (ou seja, uma plataforma abstrata) ainda seria inviável. Quando se compara, por exemplo, as referências à infraestrutura de um sistema embarcado utilizando o RTOS *Windows Embedded Compact* 6.0[4] da Microsoft com aquelas oferecidas pelo RTOS *X Real Time Kernel* da eSysTech, percebe-se a imediata inviabilidade de criação de um modelo abstrato comum. Nem o propósito, nem a estrutura e, tampouco, a lógica funcional destes RTOSs permitem uma associação direta entre eles. A mesma constatação pode ser facilmente feita com outros RTOSs, como *Linux Embedded*[5] e VxWorks.

Neste sentido, a busca por uma padronização das interfaces de sistemas operacionais já foi tentada no passado, como, por exemplo, por meio do padrão POSIX (*Portable Operating System Interface*). Trata-se de uma família de padrões definidos pela IEEE para manter a compatibilidade entre sistemas operacionais (BUTENHOF, 1997). Embora vários sistemas operacionais mantenham compatibilidade com estes padrões, como VxWorks e QNX, outros RTOS não os seguem ou seguem variações destes padrões. Ademais, como o POSIX só contempla um subconjunto das funcionalidades e interfaces dos sistemas

[4] http://www.microsoft.com/windowsembedded/en-us/evaluate/windows-embedded-compact-7.aspx
[5] http://www.uclinux.org/

operacionais (principalmente suas funções centrais ligadas ao escalonamento e criação de processos e tarefas), este padrão estaria longe de poder ser considerado uma camada de abstração completa para representar os sistemas operacionais atuais.

Nesta pesquisa de doutorado, considera-se que a necessidade de independência do modelo de um *software* embarcado (i.e., do PIM de uma aplicação embarcada) se limita satisfatoriamente a um determinado RTOS operando sobre uma variação limitada de tecnologias de *hardware*. Esta pesquisa de doutorado não investigou esta hipótese cientificamente, pois ultrapassa seu escopo. Porém, as evidências recolhidas de profissionais da área, inclusive daqueles que empregam o RTOS *X Real time Kernel*, não indicam uma necessidade explícita de modelos com independência além do que se está considerando, ou seja, um RTOS com algumas poucas variações sobre uma arquitetura de *hardware* também pouco variante. A portabilidade de uma aplicação entre RTOSs e arquiteturas de *hardware* diferentes não se apresenta como uma necessidade importante. Seguramente, esta suposição deverá ser investigada em trabalhos futuros.

Assim, considera-se nesta pesquisa que a independência necessária a um modelo de aplicação (PIM) se limita a um conjunto definido e razoavelmente homogêneo de variações do ambiente de execução, ou seja, de plataformas, envolvendo um determinado RTOS e algumas variações de tecnologias de *hardware* compatíveis. Dada a afinidade destas plataformas, utiliza-se aqui o termo “família de plataformas” para fazer referência a este conjunto. Assim, uma família de plataformas define um escopo limitado dentro do qual os modelos de aplicações (PIM) podem ter total independência de plataforma.

Desse modo, a independência de plataforma, sob a ótica do PI-MT, é suportada para uma família de plataformas por meio do uso de elementos de um perfil de plataforma e de um perfil de ligação. O perfil de plataforma define uma plataforma abstrata (composta por um RTOS e uma família de processadores associados), por sua vez, o perfil de ligação define os elementos de ligação entre o modelo da aplicação (PIM) e o modelo de plataforma (PM). O uso de perfis para a realização da transformação de modelos é fortemente incentivado no desenvolvimento de *software* baseado na abordagem MDA (LOPES *et al*., 2005; BÉZIVIN *et al*., 2006). Esses perfis (perfil de plataforma e perfil de ligação) serão detalhados na subseção apresentada a seguir.

4.4 PERFIL DE PLATAFORMA E PERFIL DE LIGAÇÃO

Um perfil de plataforma, também denominado metamodelo de plataforma, pode ser interpretado como uma linguagem para descrever uma interface para tal plataforma (THOMAS *et al.*, 2007). Esta interface não se limita à descrição estrutural dos serviços fornecidos pela plataforma, mas em adição, reúne as funcionalidades dessa plataforma observáveis de um ponto de vista do usuário. Desse modo, o modelo de plataforma gerado com base no perfil de plataforma pode ser considerado como um modelo de tal interface, bem como, o perfil de plataforma pode ser considerado como uma linguagem para descrever essa interface.

Um perfil de plataforma pode auxiliar a resolver o problema da "independência de plataforma" em *software* embarcado. No contexto dessa pesquisa, um perfil de plataforma deve especificar, de forma abstrata, um conjunto de serviços oferecido por um RTOS, possibilitando a geração de Modelos de Plataforma (PMs) específicos para uma família de plataformas (compostas por um RTOS e elementos de *hardware* específicos). Desse modo, o perfil de plataforma define elementos que permitem a representação de um conjunto de serviços típicos do RTOS, por meio da definição de uma camada de abstração que possibilite referenciar estes serviços em um modelo de aplicação.

No contexto desta pesquisa, o perfil de plataforma é utilizado como parte fundamental para o desenvolvimento de *software* embarcado, fornecendo meios de obter independência de plataforma para uma família de plataformas computacionais embarcadas. Em adição, o método proposto utiliza um perfil de ligação que define estereótipos a serem aplicados em elementos do modelo PIM, visando especificar os elementos desse modelo que utilizam os serviços da plataforma de implementação. O perfil de ligação deve permitir ao engenheiro de *software* modelar o PIM, sem exigir conhecimentos muito específicos sobre a plataforma selecionada. Dessa forma, o perfil de ligação permite anotar (marcar) os elementos do modelo da aplicação que irão utilizar os serviços da plataforma, definidos no PM. Portanto, o modelo PIM deve possuir um nível suficiente de independência de plataforma, possibilitando a sua transformação em modelos específicos de plataforma, de acordo com o RTOS selecionado e respectivo *hardware* associado, dentro de uma família de plataformas.

Os passos de aplicação do método PI-MT utilizam os conceitos de perfil de plataforma e de perfil de ligação. Esses passos serão detalhados na próxima subseção.

4.5 PASSOS DE APLICAÇÃO DO PI-MT

O método PI-MT define os seguintes passos a serem realizados para criação e uso de transformações de modelos reutilizáveis (genéricas): 1) Criação do Modelo de Transformação (MT); 2) Definição do modelo PIM; 3) Seleção do PM; 4) Execução da Transformação; 5) Verificação da Transformação (AGNERa *et al*., 2012). A sequência dos passos realizados pelo método PI-MT é ilustrada na Figura 16. A Figura 17 complementa a Figura 16 e utiliza a técnica SADT (*Structured Analysis and Design Technique*) (ROSS, 1985) para detalhar as informações necessárias para a realização de cada passo, bem como a saída produzida e as ferramentas de auxílio utilizadas.

4.5.1 PI-MT Passo 1: Criação do Modelo de Transformação

O desenvolvimento de *software* dirigido a modelos é baseado no conceito de modelos em conformidade com seus metamodelos. Dessa forma, transformações de modelos podem ser especificadas por meio de um modelo de transformação.

Um modelo de transformação pode ser definido como a descrição de um relacionamento entre os elementos do modelo-fonte e os elementos do modelo-alvo, estabelecido por meio de uma transformação. Portanto, a transformação é efetivamente definida como um modelo, denominado Modelo de Transformação (MT), o qual é baseado em um metamodelo de uma Linguagem de Transformação de Modelos (MTL, acrônimo de *Model Transformation Language*) (BÜTTNER *et al*., 2011; JOUAULT *et al*., 2008).

O Modelo de Transformação (MT) que compõe o método PI-MT contempla as seguintes características, especificadas em (CZARNECKI; HELSEN, 2006):

- **Domínio**: trata dos modelos considerados pela transformação, sendo eles: domínio de origem, composto pelos modelos PIM e PM, e domínio de destino, composto pelo modelo PSM.
- **Organização**: compreende a estruturação geral da composição das regras, que envolve o uso de mecanismos de modularização e reutilização. Esta tese utiliza a técnica de composição denominada Sobreposição de Módulos, proposta por (WAGELAAR, 2010), que possibilita a organização da transformação em módulos distintos, bem como o reuso destes módulos.

- **Metamodelos**: um domínio é associado a um metamodelo, que descreve as possíveis estruturas e construtores dos modelos para esse domínio.
- **Direcionalidade**: uma transformação de modelos pode ser executada em apenas uma direção (transformação unidirecional) ou em múltiplas direções (transformação multidirecional). No contexto desta tese, as transformações são unidirecionais (PIM-para-PSM), dado que este requisito atende as necessidades do problema abordado.

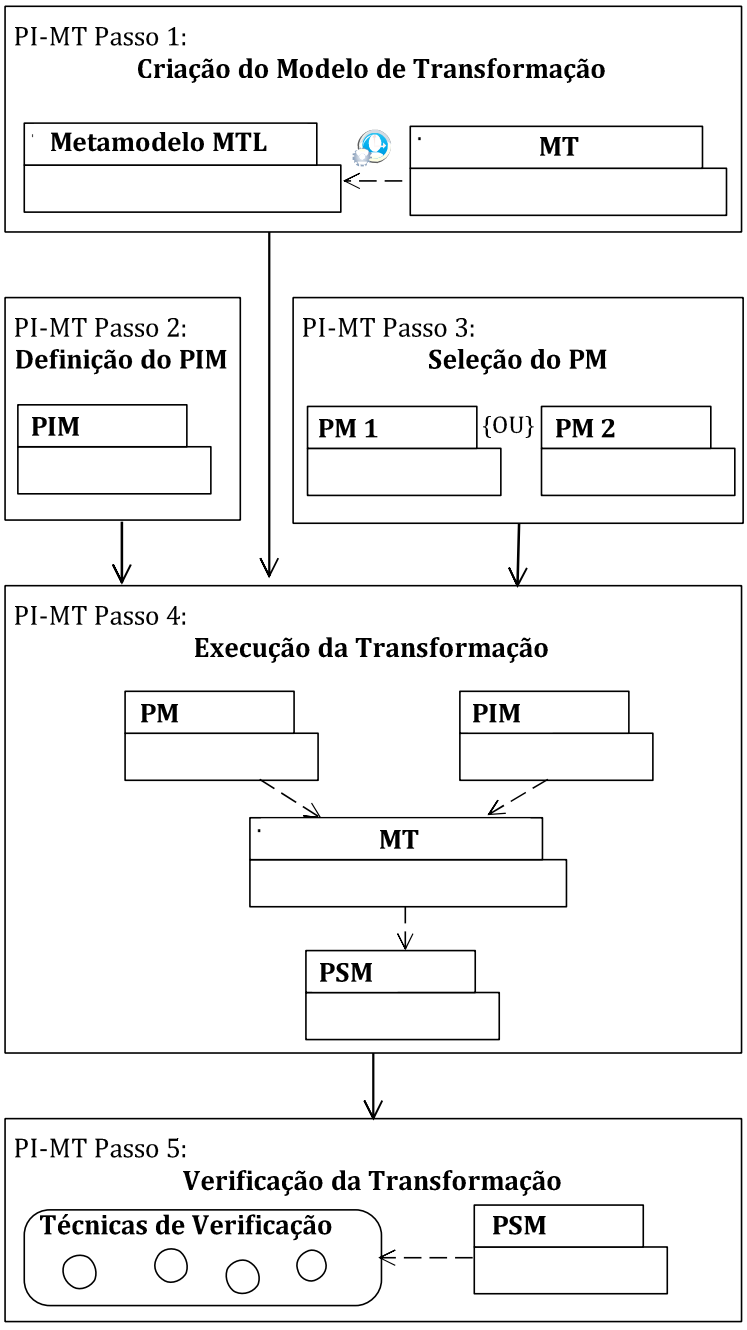

Figura 16 – Passos do método PI-MT
Fonte - Adaptada de (AGNERa *et al.*, 2012)

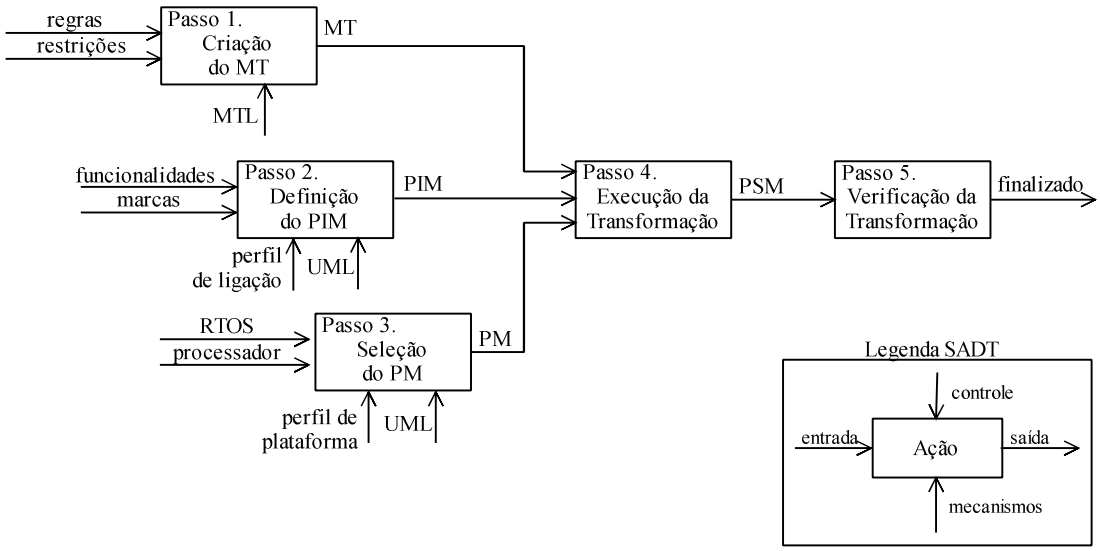

Figura 17 – Detalhamento dos passos do PI-MT

A Figura 18 ilustra uma visão geral do MT proposto nesta pesquisa. As entradas do MT são o PIM (modelo-fonte) e o PM (modelo de plataforma). Por sua vez, o modelo de saída é o PSM (modelo-alvo). O PIM-MT utiliza a metamodelagem baseada em UML. Desse modo, o PIM e o PSM devem estar em conformidade com o metamodelo UML. Atualmente, a UML é a linguagem padrão de modelagem de *software* e, portanto, desempenha papel fundamental em abordagens dirigidas e modelo (LOUHICHI *et al.*, 2011; PRASHANTH e SHET, 2009).

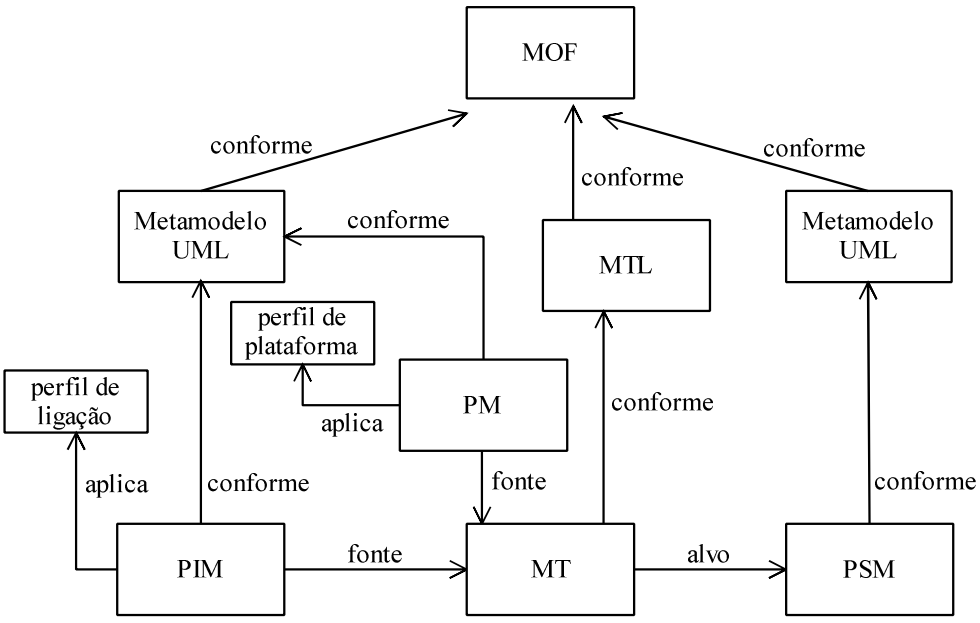

Figura 18 – Padrão do Modelo de Transformação do Método PI-MT

Transformações de modelos são classificadas como verticais quando o modelo-fonte e o modelo-alvo residem em diferentes níveis de abstração (MENS e VAN GORP, 2006). No contexto desta pesquisa, uma transformação é o refinamento dito vertical de um modelo-fonte, adicionando a este modelo detalhes de uma plataforma alvo específica. Em adição, no MT proposto os modelos de origem e de destino estão em conformidade com um metamodelo único, o metamodelo da UML. Nesse caso, a transformação proposta é classificada como endógena. Transformações endógenas são realizadas entre modelos expressos com base no mesmo metamodelo (SUN *et al*., 2009; MENS e VAN GORP, 2006).

As características específicas de plataforma são tratadas pelo MT de forma independente, por meio de PMs utilizados como modelos de entrada no processo de transformação. Desse modo, conectar as características de uma plataforma-alvo, expressa por meio de PMs, com os modelos da aplicação (PIMs), é um dos grandes desafios que precisam ser enfrentados em soluções de transformação de modelos genéricas.

Em resumo, a “independência de plataforma” é alcançada no PI-MT escondendo-se os detalhes de uma família de plataformas em nível de arquitetura de *software* do nível da aplicação, por meio de estereótipos definidos no perfil de ligação e aplicados ao modelo PIM. O Modelo de Transformação é representado por meio de um programa (ou módulo) de transformação, que especifica as regras da transformação em uma linguagem de transformação específica.

Outra importante questão na definição do MT é a criação do programa ou módulo de transformação por meio do uso de uma MTL. Desse modo, o programa de transformação define as regras que devem constituir a transformação por meio de uma linguagem como a *Atlas Transformation Language* (ATL). Por sua vez, um programa é formado por uma ou mais regras de transformação, cada qual composta pelas seguintes partes: o lado esquerdo da regra (*Left-Hand Side* - LHS) e o lado direito da regra (*Right-Hand Side* - RHS). O LHS refere-se ao modelo-fonte, enquanto o RHS cria ou expande o modelo-alvo (CZARNECKI e HELSEN, 2006). Dentro desse contexto, é importante delimitar o escopo de aplicação de uma regra, i.e., os elementos do modelo-fonte que devem disparar a execução das regras de transformação, visando a restringir as partes do modelo que irão participar da transformação.

Ainda, a MTL selecionada para realizar a transformação de modelos deve permitir a recuperação de informações de rastreabilidade, utilizadas para resolver as interações e dependências implícitas entre as diferentes regras envolvidas na transformação executada (YIE e WAGELAAR, 2009). A rastreabilidade é a habilidade de estabelecer níveis de relacionamento entre dois ou mais elementos em um processo de desenvolvimento,

especialmente para elementos que possuem um relacionamento do tipo origem-destino, como é o caso de transformações de modelos (GALVÃO e GOKNIL, 2007). Essas informações são usadas para auxiliar na interação entre diferentes regras de transformação, onde uma regra pode usar como dado de entrada a saída produzida por outras regras.

4.5.2 PI-MT Passo 2: Definição do PIM

A definição do PIM é o segundo passo a ser realizado para utilizar o método PIM-MT. Em termos gerais, este modelo deve definir as funcionalidades da aplicação e os serviços de RTOS necessários em alto nível de abstração. O PIM deve possuir um nível suficiente de independência de plataforma, permitindo sua transformação em PSMs, de acordo com a família de plataformas selecionada (RTOS e o respectivo *hardware* associado).

O modelo PIM deve aplicar em seus elementos marcas (por meio de estereótipos) que especificam os serviços de RTOS utilizados, com base nos estereótipos definidos no perfil de ligação. As informações definidas por meio destas marcas possibilitam buscar no PM as características de plataforma a serem inseridas no modelo alvo (PSM). Desse modo, é preciso considerar o custo de treinamento do desenvolvedor da aplicação com relação ao perfil de ligação (estereótipos e propriedades). É relevante ressaltar que este custo é impactante apenas em projetos iniciais baseados no PI-MT, visto que em projetos posteriores o desenvolvedor já terá adquirido os conhecimentos requeridos. Além disso, estudar os elementos do perfil de ligação é significativamente mais simples do que possuir habilidades para modelar todos os detalhes da plataforma-alvo.

O método PI-MT tem como foco os diagramas de classe e de sequência para a modelagem do PIM. Diagramas de classe definem os componentes do sistema de *software* e especificam os relacionamentos entre eles. Diagramas de sequência especificam os cenários das interações representativas entre os objetos (das classes) que constituem o sistema ou que interagem com ele (OMG, 2011a). Pesquisas realizadas apontam estes diagramas como os mais utilizados na modelagem de sistemas de *software* utilizando a UML (AGNERc *et al*., 2012; DOBING e PARSONS, 2006; GROSSMAN *et al*., 2005).

4.5.3 PI-MT Passo 3: Seleção do PM

O termo "plataforma" refere-se à combinação de *software* e *hardware* necessários para executar uma aplicação (SELIC, 2005). Por sua vez, um Modelo de Plataforma define um conjunto de conceitos que representam uma plataforma. No contexto desta pesquisa, cada plataforma faz parte de uma “família de plataformas” e consiste de um RTOS associado a um *hardware* incorporado, com base em um processador específico. Assim, uma família de plataformas consiste de diferentes plataformas baseadas em um único sistema operacional e associadas a diferentes processadores.

Especificamente, *software* embarcado baseado em RTOS beneficia-se pela utilização de um PM explicitamente definido, principalmente devido à ampla diversidade de plataformas existentes, formadas por um RTOS específico e um processador associado. O PM deve especificar o conjunto de mecanismos de *hardware* e *software* que habilitam a execução de aplicações de *software* embarcado. O *software* refere-se ao RTOS e em suas respectivas APIs e o *hardware* refere-se às plataformas de *hardware* baseadas em processadores específicos. Desse modo, dependendo da plataforma utilizada pela aplicação, um determinado PM deve ser selecionado e utilizado como modelo de entrada na execução da transformação de modelos.

No contexto desta tese, um perfil de plataforma deve definir uma camada de abstração de um RTOS de forma genérica, não considerando uma plataforma de *hardware* específica, isto é, um processador específico acoplado a uma placa eletrônica específica. Como diversas plataformas podem ser baseadas em um único RTOS, diferentes PMs pertencentes a uma mesma família de plataformas podem ser criados em conformidade com o respectivo perfil. Desse modo, esse passo do PI-MT visa selecionar um PM específico a ser utilizado como entrada do processo de transformação.

4.5.4 PI-MT Passo 4: Execução da Transformação

A execução da transformação é realizada de acordo com as regras definidas no programa de transformação e utiliza o PIM (definido no Passo 2) e o PM (selecionado no Passo 3) como modelos de entrada. O PIM representa a modelagem da aplicação e o PM representa os conceitos da plataforma. Como resultado, é gerado o PSM contendo detalhes pertencentes a uma plataforma-alvo particular.

Considerando linguagens de transformação de modelos baseadas em regras, como ATL ou Epsilon, o modelo de transformação desenvolvido no Passo 1 é compilado e executado por um mecanismo de execução (VAN AMSTEL *et al*., 2011; KOLOVOS *et al*., 2008). Nesse momento, o compilador verifica se o programa/módulo desenvolvido está sintaticamente correto com relação à linguagem utilizada. Durante a execução da transformação a correspondência dos elementos do modelo-fonte com os elementos do modelo-destino é realizada, com base nas condições especificadas nas regras de transformação. Para cada combinação encontrada, novos elementos são criados no modelo de destino.

4.5.5 PI-MT Passo 5: Verificação da Transformação

O desenvolvimento de transformações de modelos não é tarefa trivial e é propenso a erros (KÜSTER, 2004). O processo de verificação das transformações de modelos visa determinar se a implementação de uma transformação de modelos satisfaz à suas especificações (LAMARI, 2007). Nesta etapa é necessário, além de depurar as regras definidas na transformação, verificar se os modelos produzidos atendem aos requisitos especificados pela transformação.

Desse modo, os testes do programa de transformação são uma parte importante do processo de desenvolvimento de *software*. Testes são realizados para revelar defeitos em uma transformação, para assegurar que a definição da transformação está em conformidade com sua especificação e para verificar que a execução da transformação se comporta da maneira pretendida.

4.6 APLICAÇÃO DO MÉTODO PI-MT

Na abordagem MDA, os modelos podem ser representados utilizando uma visão estrutural e/ou comportamental do sistema (OMG, 2003). Por um lado, a visão estrutural define a arquitetura do sistema de *software*, sendo o diagrama de classes o mais utilizado atualmente para esse fim (AGNERc *et al*., 2012; DOBING e PARSONS, 2006; GROSSMANN *et al*., 2005). Esse diagrama descreve as classes de um sistema de *software*, incluindo suas propriedades e relacionamentos.

Por outro lado, a visão comportamental concentra-se no comportamento das classes definidas na visão estrutural. Os diagramas de sequência são apontados como os modelos comportamentais mais utilizados atualmente (DOBING e PARSONS, 2006; GROSSMANN *et al*., 2005; AGNERc *et al*., 2012) e especificam como as instâncias de elementos do modelo interagem com outros elementos.

Com base no método PI-MT proposto, esta pesquisa realizou a criação e implementação de duas transformações de modelos PIM-para-PSM, sendo elas: MT-AMP e MT-PROAPES. A primeira é voltada para a transformação de diagramas de classe e baseada no perfil de ligação AMP. Por sua vez, a transformação MT-PROAPES é voltada para a transformação de diagramas de sequência e baseada no perfil de ligação *dynRTOS*. Ambos os perfis de ligação são integrantes do perfil PROAPES, definido por Soares (2012).

O ambiente TopCased foi utilizado como o ambiente para edição, definição e execução das transformações de modelos realizadas por esta pesquisa. Esse ambiente é bastante utilizado no desenvolvimento de transformações de modelos voltadas para o projeto de sistemas embarcados utilizando abordagens dirigidas à modelo (PONTISSO e CHEMOUIL, 2006).

Nas transformações MT-AMP e MT-PROAPES foram consideradas como plataforma alvo o RTOS "*X Real-Time Kernel*" empregado em processadores ARM. O perfil PROAPES, proposto por Soares (2012), foi o perfil de plataforma usado como referência para a realização dessas transformações. Esse perfil visa à criação de PMs voltados para o núcleo operacional *X Real Time Kernel* e será apresentado sucintamente na próxima subseção.

4.7 MODELO DE PLATAFORMA

O modelo de plataforma usado como entrada das transformações de modelos implementadas nesta pesquisa é definido por meio da aplicação de elementos de um perfil de plataforma. Por sua vez, esse perfil deve especificar, de forma abstrata, um conjunto de serviços oferecido pelo RTOS *X Real-Time* Kernel.

O Perfil para a Modelagem da Aplicação e Plataforma de *Software* Embarcado, denominado *Profile for Modeling Application and Platform of Embedded Software* (PROAPES), proposto por Soares (2012), foi o perfil de plataforma utilizado como referência para criação dos PMs. Tal perfil foi definido para descrever genericamente os serviços

fornecidos por uma família de plataformas (composta pelo RTOS *X* e respectivos processadores associados).

A arquitetura do perfil PROAPES, ilustrada na Figura 19, é composta por cinco pacotes, sendo eles:

- AMP (*Application Modeling Profile*): perfil de modelagem da aplicação.
- *BasicTypesX*: biblioteca de modelos referente aos artefatos de tipos básicos do RTOS *X Real-Time Kernel*.
- *dynRTOS*: perfil de modelagem de conceitos dinâmicos, i.e., artefatos para modelar diagramas de sequência e diagramas de atividades em projeto de *software* embarcado baseado no RTOS X *Real-Time Kernel*.
- *swxRTOS*: perfil de modelagem de artefatos da plataforma, i.e., artefatos relacionados a *drivers* de dispositivos e artefatos para construções de concorrência e interações.
- *TypesProcessorX*: biblioteca de modelos relacionada aos artefatos de tipos referentes aos processadores usados pelo RTOS X *Real-Time Kernel*.

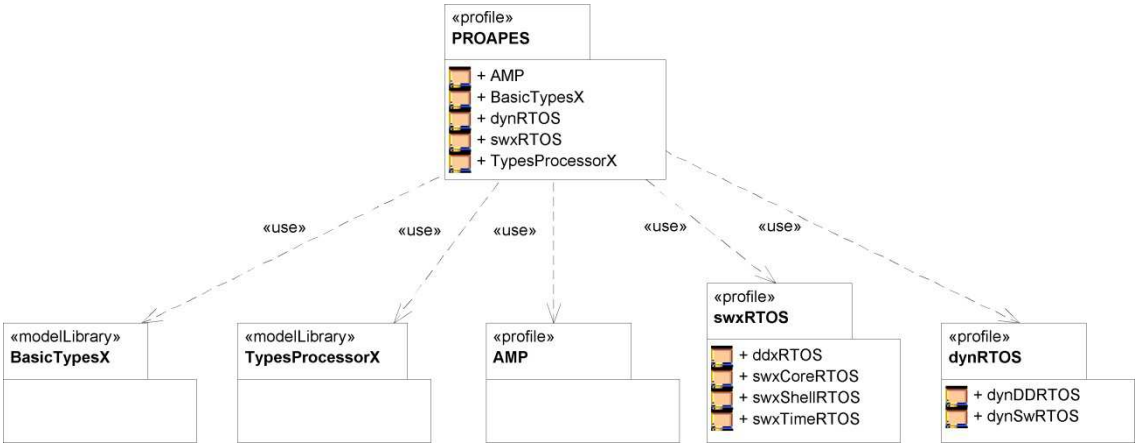

Figura 19 - Arquitetura do perfil PROAPES
Fonte: Soares (2012)

O pacote *swxRTOS* que define artefatos de modelagem para a descrição de contextos de *software* de execução concorrente para o RTOS *X Real-Time Kernel* será detalhado a seguir.

4.7.1 Perfil *swxRTOS*

O perfil *swxRTOS*, ilustrado na Figura 20, é composto pelos seguintes subperfis:

- *swxCoreRTOS*: representa os conceitos básicos de construções de alto nível que são necessários para apoiar o uso de construções de concorrência e interações.
- *swxTimeRTOS*: representa os conceitos básicos referentes a tempo.
- *swxShellRTOS*: representa os conceitos de apoio à depuração que permite ao usuário acompanhar como os recursos do processador são alocados e usados pela sua aplicação.
- *ddxRTOS:* visa o fornecimento de artefatos de modelagem para a descrição de *drivers* de dispositivos utilizados em um sistema embarcado que utilizam o RTOS *X Real-Time Kernel*.

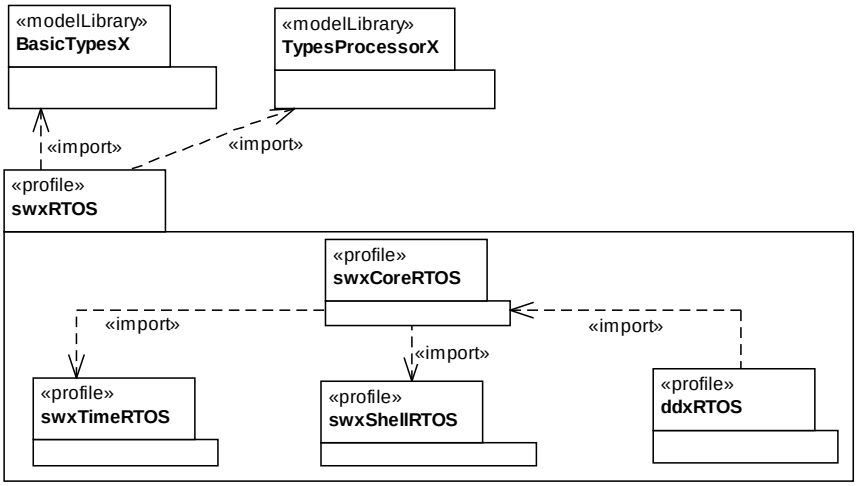

Figura 20 – Perfil *swxRTOS*
Fonte: Soares (2012)

Os subperfis que compõem o perfil *swxRTOS* importam os tipos de dados definidos na biblioteca de modelos *BasicTypesX*, bem como importam os tipos de processadores definidos na biblioteca de modelos *TypesProcessorX*.

O subperfil *swxCoreRTOS* é ilustrado na Figura 21. Esse subperfil define cinco estereótipos, sendo eles:

- *swxCore:* representa os conceitos básicos de descrição de contextos de *software* de execução concorrente para o RTOS *X Real-Time Kernel.*

- *swxSemaphore:* representa os conceitos básicos para criação e gerenciamento de um semáforo para o RTOS X *Real-Time Kernel*. Desse modo, as tarefas podem sincronizar-se mutuamente implementando as regiões críticas do RTOS *X Real-Time Kernel*.
- *swxPipe:* representa um *pipe* entre duas tarefas que armazena as referências para os objetos de dados.
- *swxISR_Pipe:* representa um *pipe* entre um ISR anônimo e uma *thread* que armazena as referências para os objetos de dados.

Para exemplificar o uso do perfil *swxRTOS* na criação de modelos de plataforma, um fragmento de um PM específico para a plataforma RTOS *X Real-Time Kernel* versão 1.0 em processadores ARM7 é apresentado na Figura 22. Este fragmento é representativo, pois refere-se ao subperfil *swxCoreRTOS* que especifica importantes serviços do *kernel* X.

O perfil *swxCoreRTOS*, ilustrado na Figura 22, está relacionado com o PM denominado pacote “xl”. As classes do pacote “xl” são marcadas com estereótipos definidos no subperfil *swxCoreRTOS*. Por exemplo, a classe “X” está anotada com o estereótipo *swxCore* que representa os conceitos básicos de descrição de contextos de *software* de execução concorrente para o RTOS *X Real-Time Kernel*. Este estereótipo representa conceitos como: programação concorrente, escalonamento de tarefas, prioridade entre tarefas, sincronização entre tarefas, comunicação entre tarefas, organização de mensagens em filas de execução, exclusão mútua e temporizadores. Desse modo, o relacionamento entre o PM e o perfil da plataforma para esses conceitos é realizado por meio do estereótipo *swxCore*.

As propriedades associadas ao *swxCore* permitem definir anotações nas classes que possuem esse estereótipo aplicado. Desta forma, as propriedades indicam os valores rotulados (*tagged-values*) associados ao estereótipo. A Figura 23 ilustra alguns valores rotulados atribuídos aos atributos e operações da classe “X”. Nessa figura, o PM possui uma classe chamada "X" que define atributos e operações descritos de acordo com a versão do *kernel* X versão 1.0 em processadores ARM7.

O estereótipo *swxCore* é aplicado na classe "X" do PM (Figura 23). Neste exemplo, as seguintes meta-propriedades do estereótipo *swxCore* são descritas: i) *threadId*: identificação da *thread*; ii) c*heckMessage*: verifica a existência de mensagens nas caixas de entrada; e iii) *getNameThread*: retorna o nome da *thread*.

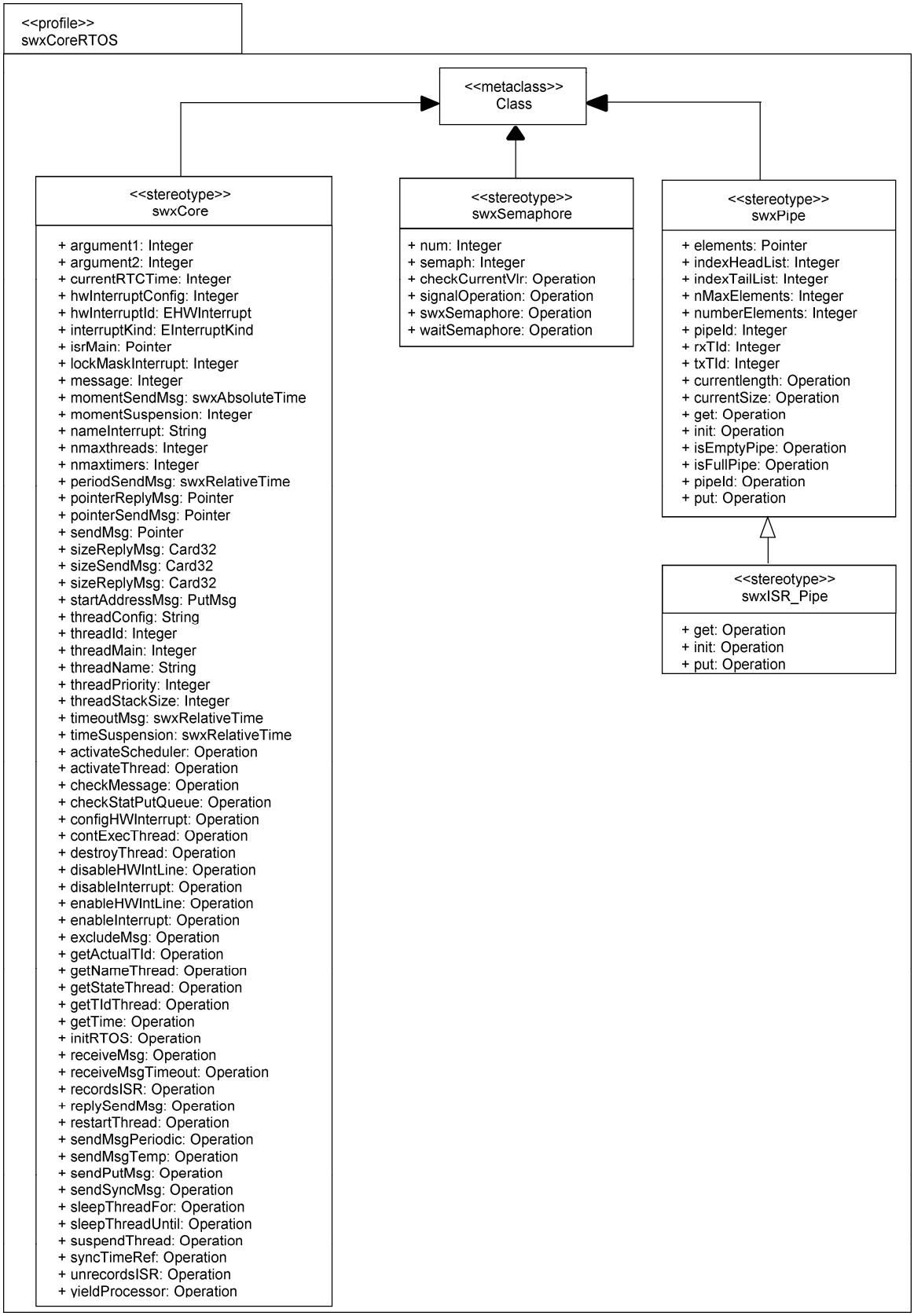

Figura 21 – Perfil *swxCoreRTOS*
Fonte: Soares (2012)

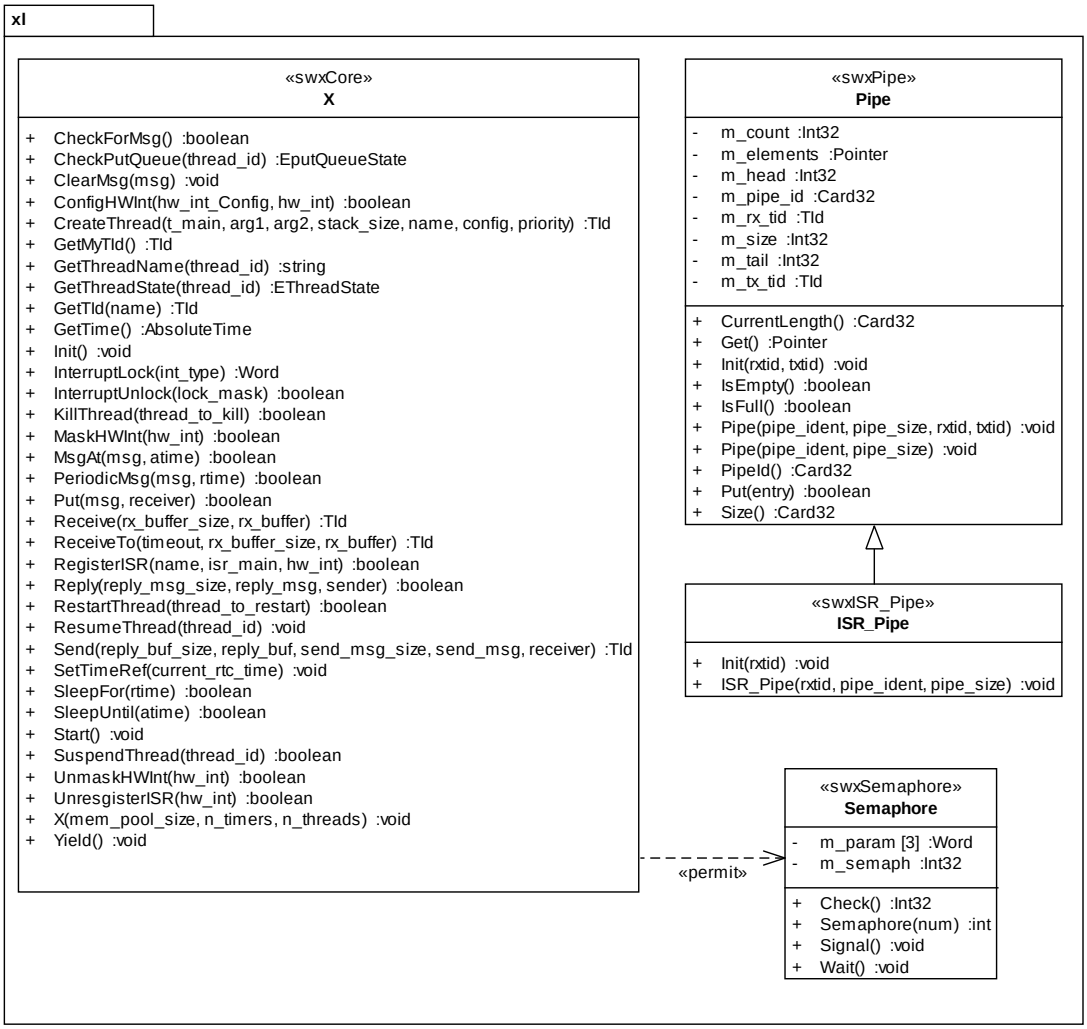

Figura 22 – Fragmento do PM – RTOS X ARM7
Fonte: Soares (2012)

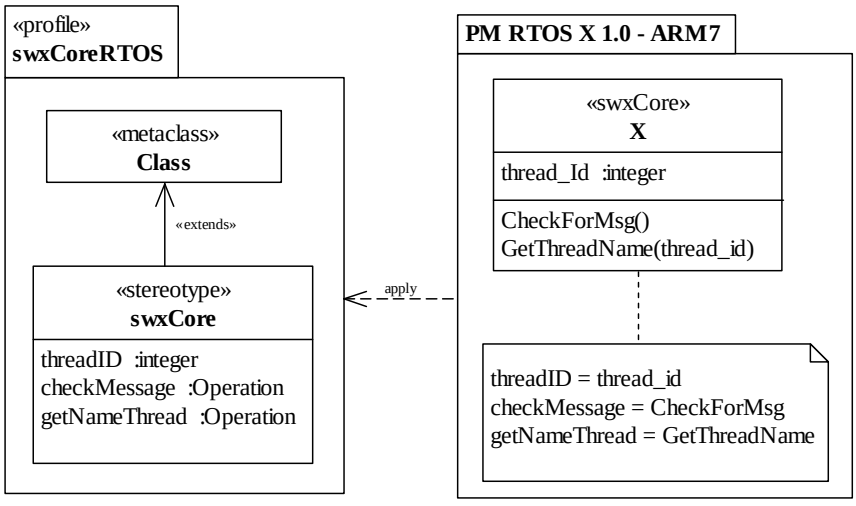

Figura 23 - PM gerado com base no perfil de plataforma

O mapeamento, representado na Figura 23, é realizado definindo-se um nível de dependência entre as meta-propriedades do perfil *swxRTOS* e as propriedades do PM. Isto acontece por meio de propriedades (valores rotulados), de acordo com as anotações relacionadas com a classe "X". Por exemplo, a propriedade *checkMessage* definida no perfil *swxCoreRTOS* é substituída pelo método *CheckForMsg*() no PM, com base nos valores rotulados associados ao estereótipo *swxCore*. Desse modo, o uso de valores rotulados adiciona semântica aos elementos do PM.

4.8 TRANSFORMAÇÃO DE MODELOS MT-AMP

A transformação de modelos MT-AMP (*Model Transformation - Application Modeling Profile*), integrante do método PI-MT, visa prover independência de plataforma em transformações de modelos PIM para PSM, concentrando-se no desenvolvimento de *software* embarcado que faz uso de RTOS. A MT-AMP tem como foco a modelagem estática do sistema de *software* e suporta o uso de diagramas de classes da UML.

Uma visão geral da transformação MT-AMP, proposta nesta tese, é apresentada na Figura 24. Nessa figura, os retângulos coloridos representam o fluxo da transformação por meio dos modelos de entrada (PM e PIM), modelo de saída (PSM) e modelos de transformação (PIM2PSM.atl e UML2Copy.atl). Os demais retângulos representam os metamodelos usados na definição desses modelos (metamodelo UML, metamodelo ATL, MOF, perfil *swxRTOS* e perfil AMP).

Conforme ilustrado na Figura 24, os modelos de entrada da transformação são o PIM e o PM, enquanto que o modelo de saída é o PSM. O modelo PIM aplica elementos do perfil AMP (*Application Modeling Profile*), enquanto o PM aplica elementos do perfil *swxRTOS*. A MT-AMP é dirigida pelo módulo de transformação denominado PIM2PSM.atl que contém as regras específicas que realizam as mudanças necessárias nos elementos do modelo de origem relacionados com os serviços de RTOS, ou seja, esse módulo PIM2PSM.atl contém as regras do refinamento em si. O módulo PIM2PSM.atl sobrepõe (ou superpõe) o módulo UML2Copy.atl por meio da técnica de *sobreposição de módulos* (*module superimposition*). O módulo UML2Copy.atl, proposto por WAGELAAR *et al*. (2010) e publicamente disponibilizado[6], realiza a cópia de todos os elementos de um modelo UML para outro.

[6] http://ssel.vub.ac.be/ssel/research:mdd:casestudies

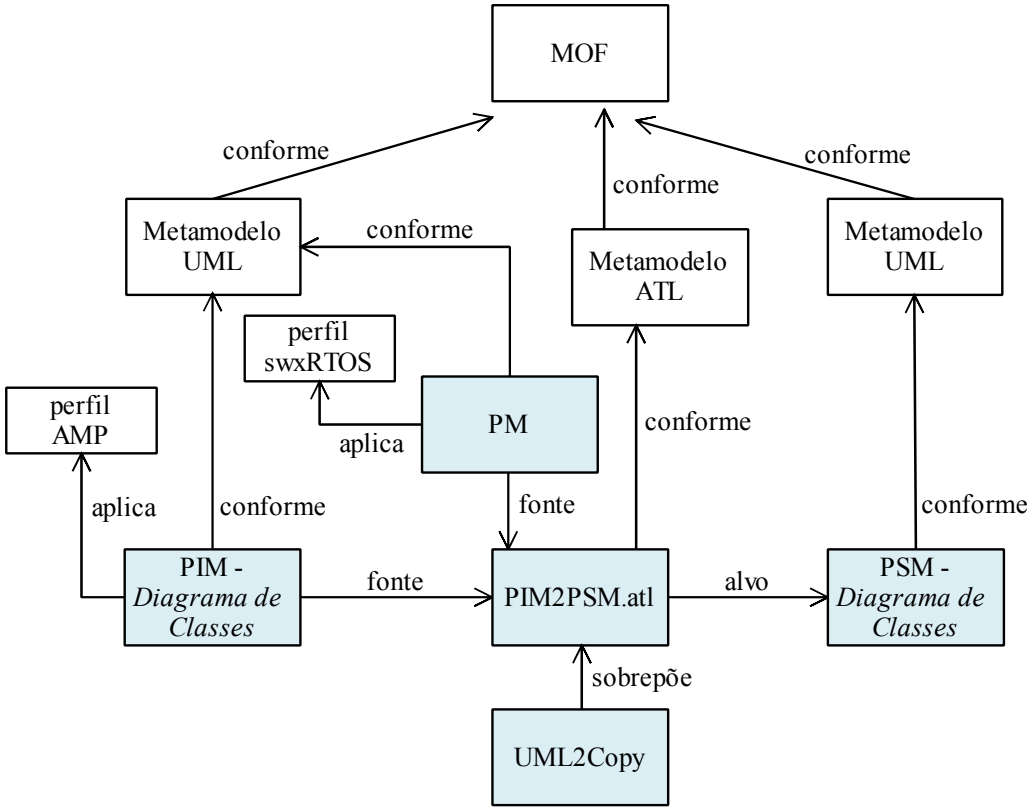

Figura 24 - Visão Geral da MT-AMP

A MT-AMP foi desenvolvida no ambiente TopCased baseado no Eclipse. A configuração da técnica de *sobreposição de módulos* é ilustrada nas Figuras 25 e 26. A Figura 25 apresenta a configuração da transformação, dirigida pelo módulo UML2Copy.atl, que define os modelos de entrada (PIM.uml e PM.uml) e o modelo de saída (PSM.uml). Para todo elemento UML do modelo-fonte, o módulo UML2Copy.atl realiza a cópia deste elemento para o modelo-alvo. Desse modo, cada regra de transformação que compõe o módulo UML2Copy.atl é responsável por realizar a cópia de um elemento específico do metamodelo UML. Por sua vez, a configuração da sobreposição do módulo PIM2PSM.atl sobre o módulo UML2Copy.atl é realizada por meio da guia “Avançada” e ilustrada na Figura 26. O módulo PIM2PSM.atl é adicionado à transformação base quando este é sobreposto ao módulo UML2Copy.atl.

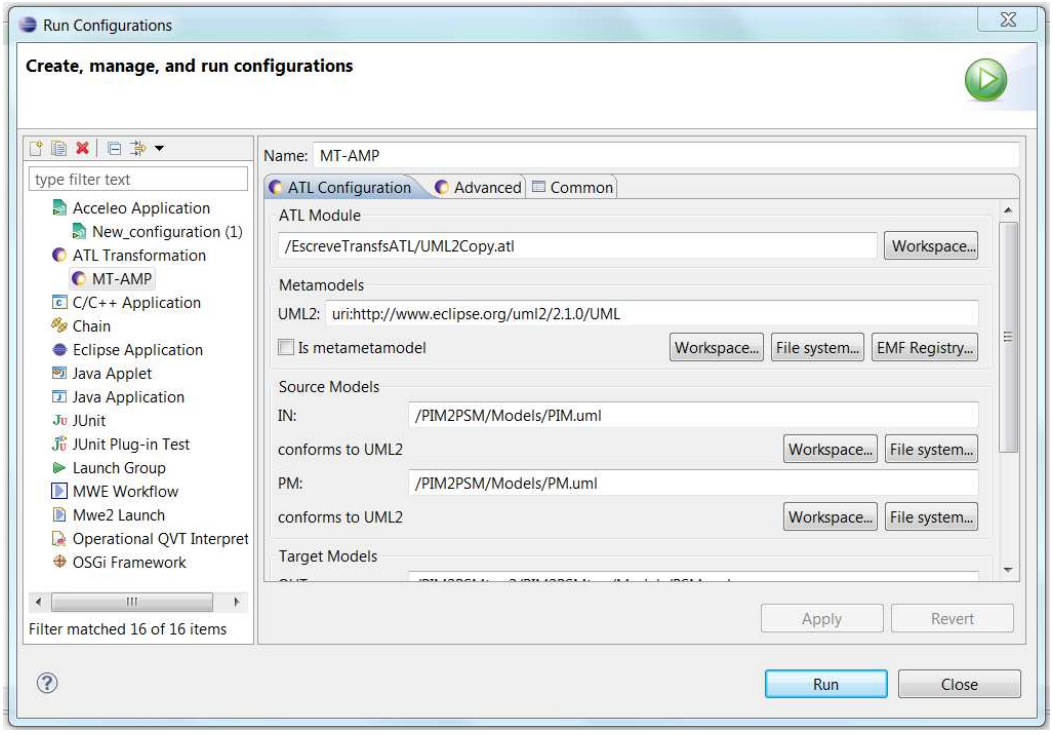

Figura 25 - Tela de configuração da transformação MT-AMP

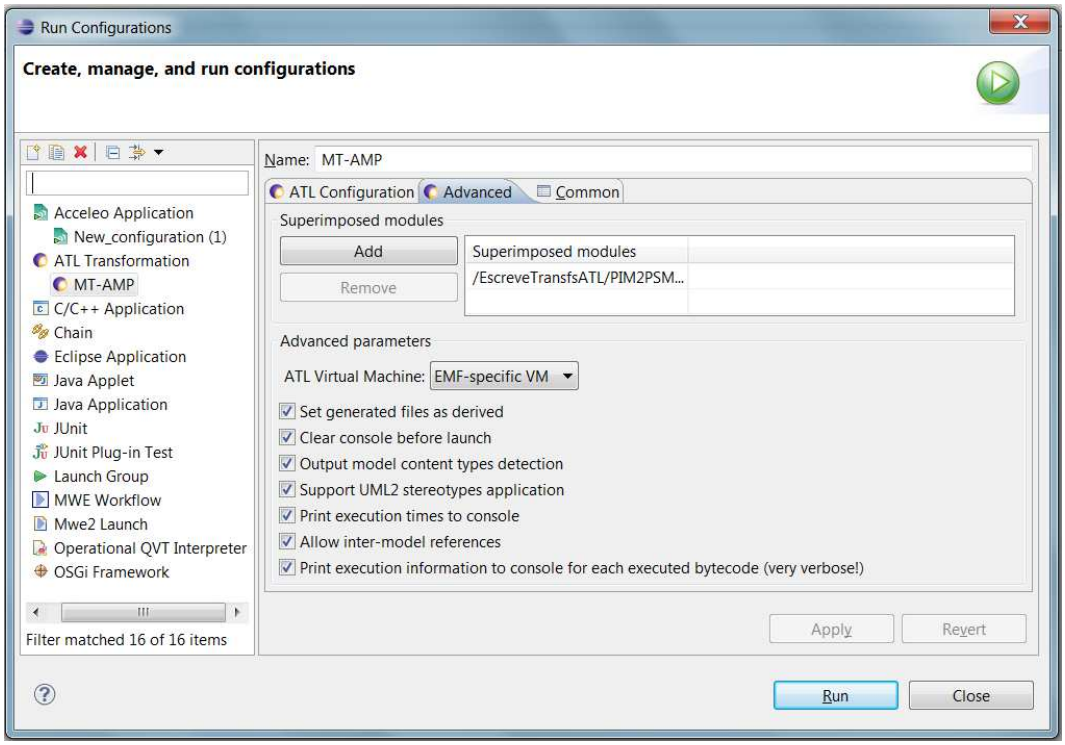

Figura 26 - Tela de configuração da sobreposição do módulo MT-AMP

A plataforma alvo de realização desta transformação é baseada no RTOS *X Real-Time Kernel* empregado em processadores ARM. O perfil de plataforma denominado PROAPES visa a criação de PMs voltados para o RTOS X e foi utilizado como referência para a realização da transformação MT-AMP. Por sua vez, o perfil de ligação responsável por realizar a ligação do PIM com o PM foi o perfil AMP (SOARES, 2012).

4.8.1 Perfil AMP

O Perfil de Modelagem de Aplicação (*Application Modeling Profile* - AMP) é utilizado pela transformação MT-AMP para a marcação de modelos PIMs. Esse perfil define um conjunto de estereótipos que podem ser utilizados para marcar operações de um modelo PIM, representado por meio de um diagrama de classes. Cada estereótipo representa um conjunto de serviços do RTOS X *Real-Time Kernel* e, desse modo, com a estrutura de estereótipos definida no AMP é possível construir um PIM com um nível suficiente de independência de plataforma, possibilitando a sua transformação em modelos específicos de plataforma, de acordo com o RTOS selecionado e respectivo *hardware* associado. O perfil AMP faz parte do perfil PROAPES.

A Figura 27 ilustra o perfil AMP e apresenta o estereótipo *ServiceRTOS* definido por meio da extensão da metaclasse *Operation*. Este estereótipo define as seguintes propriedades (*tagged-values*):

- *rtService* - armazena o nome do estereótipo a ser pesquisado no modelo de plataforma;
- *opTarget* - armazena o nome das operações de RTOS utilizadas pelos elementos do PIM.

Por sua vez, os estereótipos *rtSwOperation* e *rtDDOperation* generalizam o estereótipo *ServiceRTOS* e são utilizados para marcar os elementos do PIM, de modo a identificar os serviços de RTOS utilizados pela aplicação de forma abstrata. Desse modo, um serviço de RTOS pode ser uma operação de serviço de *software* para um RTOS, definida por meio de estereótipo *rtSwOperation*, ou uma operação de serviço de *drivers* de dispositivo, definida por meio de estereótipo *rtDDOperation*.

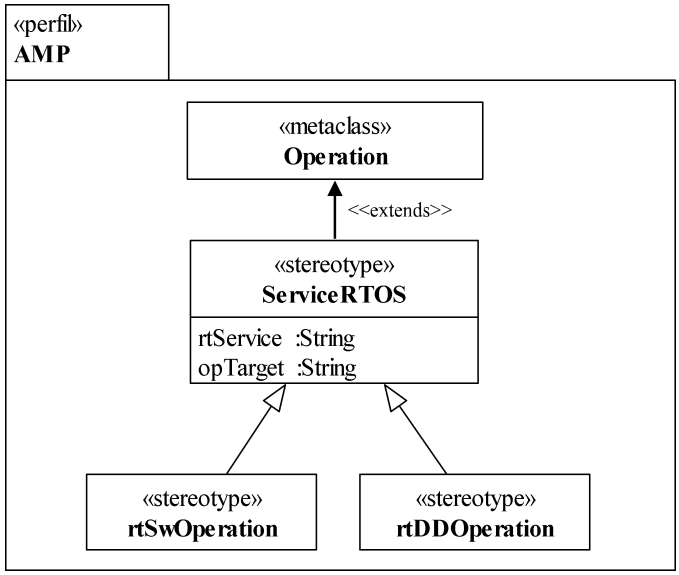

Figura 27 - Perfil AMP

4.8.2 Implementação da Transformação MT-AMP

A transformação de modelos MT-AMP foi definida por meio da linguagem ATL. O uso desta linguagem é adequado para o desenvolvimento desta pesquisa dado que a ATL provê os mecanismos necessários para produzir um modelo alvo a partir de um conjunto de modelos de entrada.

O modelo de transformação MT-AMP deve substituir os serviços genéricos de RTOS definidos no PIM por operações específicas de plataforma (apontadas no PM). Desta forma, quando a transformação é realizada, os serviços definidos de acordo com o perfil *swxRTOS* são substituídos pelos serviços definidos em um PM específico. Para tal, o modelo de transformação deve definir uma interface entre o PIM e o PM, visando identificar a forma como os elementos do PIM devem se relacionar com elementos do PM para gerar um modelo PSM com características de uma plataforma específica. A fim de estabelecer tal ligação, a transformação utiliza os estereótipos *rtSwOperation* e *rtDDOperation* definidos no perfil AMP. Estes estereótipos devem ser aplicados aos elementos do modelo PIM, de modo a identificar os elementos que utilizam os serviços de *software* e de *drivers* de dispositivo do RTOS de forma (consideravelmente) abstrata. As regras do MT devem buscar no PM pelos elementos marcados no PIM e, então, criá-los no PSM. A Figura 28 apresenta estes passos (realizados pelo modelo de transformação) e sucintamente descritos em linguagem natural.

Ø **PARA TODA** operação existente no modelo PIM **FAÇA:**
- Ø **SE** a operação estiver marcada com o estereótipo <<rtSwOperation>> **ENTÃO**:
  - Ø Recupere o valor da propriedade *rtService*.
  - Ø Procure no PM pela classe que possua o estereótipo, cujo nome combine com o valor da propriedade *rtService*, relacionada com um serviço de *software* de RTOS.
  - Ø Crie uma nova classe no modelo PSM, com o mesmo nome da classe encontrada no PM.
  - Ø Recupere as operações definidas na propriedade *opTarget*, e para cada operação existente crie uma respectiva operação no PSM.
  - Ø Crie uma associação entre a nova classe do PSM e a correspondente classe do PIM.
- Ø **SENÃO:**
  - Ø **SE** a operação estiver marcada com o estereótipo <<rtDDOperation>> **ENTÃO**:
    - Ø Recupere o valor da propriedade *rtService*.
    - Ø Procure no PM pela classe que possua o estereótipo, cujo nome combine com o valor da propriedade *rtService*, relacionada com um serviço de *driver* de dispositivo de RTOS.
    - Ø Crie uma nova classe no modelo PSM, com o mesmo nome da classe encontrada no PM.
    - Ø Recupere as operações definidas na propriedade *opTarget*, e para cada operação existente crie uma respectiva operação no PSM.
    - Ø Crie uma associação entre a nova classe do PSM e a correspondente classe do PIM.
  - Ø **SENÃO:**
    - Ø Realize uma cópia comum de tal operação.

Figura 28 – Passos realizados pela transformação MT-AMP

O modelo de transformação, implementado na linguagem ATL, resultou no módulo PIM2PSM.atl. Este módulo contém uma seção de *header* (obrigatória), *helpers* e regras de transformação que serão detalhados nas próximas subseções.

4.8.2.1 Seção header

A seção *header* define o escopo do módulo PIM2PSM.atl e declara que o modelo-fonte e o modelo-alvo devem estar em conformidade com o metamodelo UML2 (Figura 29). O *header* define os modelos utilizados como entrada da transformação, sendo eles: i) o modelo IN - refere-se ao modelo fonte (PIM); ii) o modelo PM - refere-se ao modelo de plataforma. O modelo de saída definido na seção *header*, denominado OUT, refere-se ao PSM e é gerado como resultado da transformação. Em ATL, os modelos são vinculados aos respectivos metamodelos em tempo de execução.

```
module PIM2PSM; --superimposed on UML2Copy
create OUT : UML2 from IN : UML2, PM: UML2;
```

Figura 29 - Seção *header* do módulo PIM2PSM.atl

4.8.2.2 Helpers do Módulo MT-AMP

A linguagem ATL suporta a definição de operações customizadas por meio da definição de *helpers*, expressos de acordo com a especificação da OCL (*Object Constraint Language*). Um *helper* ATL visa realizar a navegação em elementos do modelo-fonte, com o objetivo de realizar consultas ou alterações nestes elementos. O modelo de transformação aqui proposto é baseado em um perfil UML, desse modo, considerar os principais impactos causados por estereótipos e propriedades (*tagged values*) nesta implementação é fundamental, sendo eles: i) verificar se determinado estereótipo está aplicado a um elemento do modelo; e ii) consultar o valor de uma propriedade relacionada com um estereótipo específico.

Como consequência, o módulo PIM2PSM.atl define *helpers* (apresentados na Tabela 4) que verificam se estereótipos específicos são aplicados em um elemento do modelo, recuperam o valor de uma propriedade a partir de determinado estereótipo ou buscam uma determinada classe em um dado modelo.

Tabela 4 - *Helpers* do módulo PIM2PSM.atl

Nome do *helper*	**Descrição do *helper***
getTagVal	Retorna o valor de uma propriedade (*tagged value*) associada a um estereótipo específico.
getPMClass	Busca no PM por uma classe que possua um dado estereótipo aplicado. Recebe como parâmetro o nome do estereótipo pertencente ao subperfil *swxRTOS* a ser localizado em uma classe do PM.
isStereotype	Verifica se uma *Operação* do modelo-fonte possui um estereótipo aplicado.
isrtSwOperation	Verifica se uma *Operação* do modelo-fonte possui o estereótipo *rtSwOperation* aplicado.
isrtDDOperation	Verifica se uma *Operação* do modelo-fonte possui o estereótipo *rtDDOperation* aplicado.

A Figura 30 ilustra o *helper* “isStereotype” que verifica se uma operação do modelo PIM possui um estereótipo aplicado. Por sua vez, a Figura 31 ilustra o *helper* “isrtSwOperation” que verifica se o estereótipo *rtSwOperation* está aplicado a uma operação específica do modelo PIM.

```
helper context UML2!Operation def : isStereotype() : Boolean =
    self.getAppliedStereotypes()-> notEmpty();
```

Figura 30 – Helper "isStereotype"

```
helper context UML2!Operation def : isrtSwOpeartion() : Boolean =
    self.getAppliedStereotypes()->exists(st|st.name = 'rtSwOperation');
```

Figura 31 - *Helper* "isrtSwOperation"

Por fim, a Figura 32 mostra o *helper* "getTagVal" que recupera o valor de uma propriedade (*tagged value*) associada a um estereótipo específico e recebe como parâmetros o nome da propriedade e o nome do estereótipo.

```
helper context UML2!Element def : getTagVal(sName : String, pName : String ) : String =
    self.getValue(self.getAppliedStereotypes() -> select(e | e.name = sName).first(), pName);
```

Figura 32 - *Helper* "getTagVal"

4.8.2.3 Regras do Módulo MT-AMP

No escopo da linguagem ATL, a geração dos elementos do modelo alvo é alcançada por meio da definição de regras de transformação. A Tabela 5 apresenta as regras definidas no módulo PIM2PSM.atl e detalhadas a seguir.

Tabela 5 - Regras do módulo PIM2PSM.atl

Nome da regra	**Descrição da regra**
Operation	Copia as operações não marcadas do modelo PIM para o modelo PSM.
OperationStereotype	Localiza as operações do modelo-fonte marcadas com um estereótipo.
OperationRtSw	Localiza as operações do modelo-fonte marcadas com um estereótipo <<rtSwOperation>> e com base nas informações das propriedades do estereótipo realiza ações específicas para inserir características de plataforma referentes a serviços de *software* de RTOS no PSM.
OperationRtDD	Localiza as operações do modelo-fonte marcadas com um estereótipo <<rtDDOperationt>> e com base nas informações das propriedades do estereótipo realiza ações específicas para inserir características de plataforma referentes a serviços de *device drivers* de RTOS no PSM.
Model	Copia as propriedades do modelo PIM para o modelo PSM.

A regra "Operation", ilustrada na Figura 33, copia todas as operações do modelo-fonte (PIM) não marcadas pelos estereótipos *rtSwOperation* e *rtDDOperation* para o modelo-

alvo (PSM). Esta regra sobrepõe a regra de mesmo nome definida no módulo UML2Copy e será detalhada a seguir.

```
rule Operation {
    from s : UML2!Operation (
      if thisModule.inElements->includes(s) and not s.isStereotype()
      then
                s->oclIsTypeOf(UML2!Operation)
      else false endif)
      to t : UML2!Operation mapsTo s (
         name <- s.name,
         visibility <- s.visibility,
         isLeaf <- s.isLeaf,
         isStatic <- s.isStatic,
         isAbstract <- s.isAbstract,
         concurrency <- s.concurrency,
         isQuery <- s.isQuery,
         eAnnotations <- s.eAnnotations,
         ownedComment <- s.ownedComment,
         clientDependency <- s.clientDependency,
         nameExpression <- s.nameExpression,
         elementImport <- s.elementImport,
         packageImport <- s.packageImport,
         ownedRule <- s.ownedRule,
         ownedParameter <- s.ownedParameter,
         method <- s.method,
         raisedException <- s.raisedException,
         ownedParameterSet <- s.ownedParameterSet,
         templateParameter <- s.templateParameter,
         templateBinding <- s.templateBinding,
         ownedTemplateSignature <- s.ownedTemplateSignature,
         precondition <- s.precondition,
         postcondition <- s.postcondition,
         redefinedOperation <- s.redefinedOperation,
         bodyCondition <- s.bodyCondition)
}
```

Figura 33 – Regra “Operation”

Regras ATL são divididas em uma parte “from” e uma parte “to”. A parte “from” da regra especifica quais elementos do modelo de entrada (PIM) acionam a execução da regra. Por sua vez, a parte “to” da regra cria um ou mais elementos no modelo na saída (OUT).

Todas as instâncias da metaclasse *Operation* do metamodelo UML2 existentes no modelo PIM disparam a execução da regra “Operation” (Figura 33), ou seja, todos os elementos do modelo que satisfazem a condição "s->oclIsTypeOf (UML2!Operation)". Desse modo, a parte “from” seleciona no modelo-fonte apenas os elementos UML do tipo “Operation” e definidos em conformidade com o metamodelo UML2. Segundo o OMG “*uma*

operação pertence a uma classe e pode ser invocada no contexto de objetos que são instâncias de classe" (OMG, 2011a).

Na parte "from" de uma regra ATL também é possível adicionar restrições. A regra "Operation" considera apenas as operações do modelo-fonte não marcadas por estereótipos, por meio da seguinte definição de restrição "not s.isStereotype()", sendo "s" o elemento do modelo-fonte acessado pela regra.

A parte "to" da regra "Operation", ilustrada na Figura 33, cria elementos do tipo "UML2!Operation" no modelo PSM, ou seja, copia uma operação definida no modelo-fonte para o modelo-alvo. A linguagem ATL utiliza o conector '<-' para especificar uma atribuição. Considerando a regra "Operation", a cópia da operação será realizada com base nos valores das propriedades (nome, visibilidade, dentre outras) especificados no elemento do modelo original. Todas estas propriedades estão definidas na metaclasse *Operation* do metamodelo UML (OMG, 2011a).

A regra "OperationRtSw", ilustrada na Figura 34, realiza uma busca no modelo PIM por operações anotadas com o estereótipo *rtSwOperation*. Essa regra realiza as seguintes ações quando uma operação marcada com o estereótipo *rtSwOperation* é encontrada:

- A operação é copiada para o modelo alvo.
- Uma nova classe é criada no PSM, com base na classe do PM correspondente, cujo estereótipo combina com a propriedade *rtService*.
- Para toda operação de RTOS especificada na propriedade *opTarget* é gerada no PSM uma operação específica de plataforma com base no valor da propriedade opTarget.
- Uma associação é criada entre a nova classe criada no PSM e a classe que possui a operação que disparou a execução da regra.

Além disso, a regra "OperationRtSw" faz uso dos *helpers* "getPMClass" and "getTagVal". O *helper* "getPMClass" busca no PM pela classe marcada com o estereótipo apontado como parâmetro, e o *helper* "getTagVal" recupera o valor de uma propriedade associada com o estereótipo aplicado no modelo PIM. Por sua vez, a regra "OperationRtDD" tem como foco os serviços de *drivers* de dispositivos a serem inseridos no PSM e funciona de forma similar à regra "OperationRtSw".

```
rule OperationRtSw {
    from s : UML2!Operation (
    if thisModule.inElements->includes(s) and s.isrtSwOperation()
    then
       s->oclIsTypeOf(UML2!Operation)
    else false endif)
    using{
       ClassName: String =  s.getOwner().getName();
    }
    to t : UML2!Operation mapsTo s (
          name <- s.name,
          visibility <- s.visibility,
          isLeaf <- s.isLeaf,
          isStatic <- s.isStatic,
          isAbstract <- s.isAbstract,
          concurrency <- s.concurrency,
          isQuery <- s.isQuery,
          eAnnotations <- s.eAnnotations,
          ownedComment <- s.ownedComment,
          clientDependency <- s.clientDependency,
          nameExpression <- s.nameExpression,
          elementImport <- s.elementImport,
          packageImport <- s.packageImport,
          ownedRule <- s.ownedRule,
          ownedParameter <- s.ownedParameter,
          method <- s.method,
          raisedException <- s.raisedException,
          ownedParameterSet <- s.ownedParameterSet,
          templateParameter <- s.templateParameter,
          templateBinding <- s.templateBinding,
          ownedTemplateSignature <- s.ownedTemplateSignature,
          precondition <- s.precondition,
          postcondition <- s.postcondition,
          redefinedOperation <- s.redefinedOperation,
          bodyCondition <- s.bodyCondition
    ),

    newClass: UML2!Class(
        name <- thisModule.getPMClass(s.getTagVal('rtSwOperation', 'rtService')).name,
        ownedOperation <- newOp),
    newOp: UML2!Operation (name <- thisModule.getPMClass(s.getTagVal('rtSwOperation',
          ' rtService')).getTagVal(s.getTagVal('rtSwOperation', 'rtService'),s.getTagVal('rtSwOperation',
          'opTarget1')).name),
    dst : UML2!Property(
          name<-'dst',
          type<-newClass,
          aggregation <- #shared),
    src : UML2!Property(
          name<-'src',
          type<-s.getOwner()),
    newAss : UML2!"uml::Association"(
          ownedEnd <- dst,
          ownedEnd <- src )
}
```

Figura 34 - Regra “OperationRtSw”

A regra “Model”, ilustrada na Figura 35, realiza a cópia das propriedades do modelo-fonte (PIM) para o modelo-alvo (PSM). Outra importante função realizada por essa regra é a inserção no modelo PSM dos novos elementos (classes e associações) criados por meio das regras “OperationRtSw” e “OperationRtDD”. Para inserir os novos elementos no modelo PSM é preciso recuperar o *link* desses novos elementos que foram criados por outras regras. Para tal, a regra “Model” utiliza o método *resolveTemp*() que visa resolver problemas relacionados com rastreabilidade. Esse método será sucintamente explicado a seguir.

Atualmente, a linguagem ATL oferece acesso às informações referentes a rastreabilidade da transformação de modelos executada por meio do método *resolveTemp*(). As informações de rastreamento consistem da relação entre cada elemento de origem e o correspondente elemento de destino. Desta forma, estas relações são representadas como ligações de rastreamento, utilizadas pela máquina virtual ATL (ATL-VM) para resolver as interações e dependências implícitas entre as diferentes regras envolvidas na transformação executada (YIE e WAGELAAR, 2009).

```
rule Model {
   from s : UML2!Model (thisModule.inElements->includes(s))
   to t : UML2!Model mapsTo s (
      name <- s.name,
      visibility <- s.visibility,
      viewpoint <- s.viewpoint,
      eAnnotations <- s.eAnnotations,
      ownedComment <- s.ownedComment,
      clientDependency <- s.clientDependency,
      nameExpression <- s.nameExpression,
      elementImport <- s.elementImport,
      packageImport <- s.packageImport,
      ownedRule <- s.ownedRule,
      templateParameter <- s.templateParameter,
      templateBinding <- s.templateBinding,
      ownedTemplateSignature <- s.ownedTemplateSignature,
      packageMerge <- s.packageMerge,
      packagedElement <- s.packagedElement,
      packagedElement <- Set {  s.packagedElement,
         UML2!Operation.allInstancesFrom('IN')->collect(e |  thisModule.resolveTemp(e, 'newClass')),
         UML2!Operation.allInstancesFrom('IN')->collect(e | thisModule.resolveTemp(e, 'newAss'))
      },
      profileApplication <- s.profileApplication)
}
```

Figura 35 - Regra “Model”

Cada vez que uma regra de transformação é executada, a ATL define uma nova ligação entre o elemento de origem e os respectivos novos elementos gerados. Posteriormente, quando outra regra de transformação necessita implicitamente destes elementos-alvo produzidos por uma regra diferente, a ATL-VM (ATL *Virtual Machine*) resolve esta dependência utilizando as ligações de rastreamento. O método *resolveTemp*() é utilizado em módulos ATL quando uma regra necessita acessar um elemento gerado por outra regra.

Informações de rastreabilidade são parte fundamental em uma transformação de modelos, visto que são usadas para auxiliar na interação entre diferentes regras de transformação. Em ATL, cada regra pode usar a saída de outras regras por meio de um mecanismo de rastreamento implícito, uma vez que a ATL-VM oculta o acesso direto do usuário às informações de rastreamento. Assim, a regra “Model” utiliza o método *resolveTemp*() com o objetivo de inserir no modelo alvo as novas classes geradas pelas regras “OperationRtSw” e “OperationRtDD”. Vale ressaltar que, sem o uso do método *resolveTemp*() nesta transformação, as novas classes criadas seriam inseridas fora do modelo alvo (PSM). O mesmo método é utilizado para resolver a dependência das novas associações criadas no modelo alvo e relacionadas com os elementos já existentes do PIM.

4.8.3 Exemplo de Aplicação da Transformação MT-AMP

A fim de ilustrar a transformação MT-AMP, um exemplo simplificado é ilustrado na Figura 36. O exemplo realiza a transformação com base no PM RTOS X - eAt55 que considera a versão 1.0 do *X Real-Time Kernel* em processadores ARM7. Os diagramas apresentados de forma simplificada neste exemplo representam um sistema responsável por apresentar mensagens no *display*. Um diagrama de classes apresenta o fragmento de um modelo PIM referente ao módulo de controle de mensagens de um aplicativo (Figura 36). Neste exemplo, o modelo PIM contém duas classes: “CtrlMsg” e “CDisplay”. A classe “CtrlMsg” controla o fluxo de mensagens enviadas ou recebidas por processos do RTOS em execução. No exemplo, apenas a operação “enviarMsg” é definida na classe “CtrlMsg”, sendo esta responsável pelo envio de mensagens para processos do RTOS em execução. Por sua vez, a classe “CDisplay” controla um monitor conectado ao processador ARM7 e, neste exemplo simplificado, define apenas as operações “limpar” e “mostrarMsg”, sendo a primeira responsável por limpar a tela e a última por apresentar mensagens na tela.

O perfil *swxRTOS*, ilustrado de forma reduzida ao lado do modelo PIM, é utilizado para representar uma camada de abstração (perfil de plataforma) para o RTOS *X Real-Time Kernel* versão 1.0. A ligação entre o PIM e a plataforma é feita através de pontos de interação, representados por meio dos estereótipos *rtSwOperation* e *rtDDOperation*.

Desse modo, a transformação busca por elementos marcados por estes estereótipos no modelo PIM e, com base em informações definidas por meio de propriedades (valores rotulados), pesquisa no PM pelos elementos a serem inseridos no modelo alvo correspondente (PSM). Desta forma, uma nova classe é criada no PSM para suprir os serviços apontados no PIM.

Neste exemplo, a operação “enviarMsg” da Classe “CtrlMsg” está ligada à operação “sendPutMsg” do perfil *swxRTOS* por meio de valores de propriedades associadas com o estereótipo *rtSwOperation*. O estereótipo *rtSwOperation* define as seguintes propriedades: *rtService* e *opTarget*. O *rtService* indica o estereótipo a ser pesquisado no modelo PM, neste exemplo, o estereótipo *swxCore*. Por sua vez, o *opTarget* armazena os serviços de RTOS apontados no PIM com base em elementos do perfil *swxRTOS*, neste caso a operação “sendPutMsg”. O mesmo acontece com a operação “mostrarMsg” marcada com o estereótipo *rtDDOperation*, definida na classe “CDisplay” do PIM e associada com o serviço RTOS “wriStrLCD” por meio da propriedade *opTarget*.

A transformação, sucintamente ilustrada na Figura 36, tem como base o seguinte PM - eAt55: RTOS X *Real-Time Kernel* (versão 1.0) acoplado a um processador ARM7. Desta forma, o exemplo de transformação mostra a geração de um modelo PSM baseado nesta plataforma de referência. O estereótipo *swxCore* (definido no perfil *swxRTOS*) é aplicado à classe “X” do PM, bem como o estereótipo *ddxLCD* é aplicado à classe “CDDX_LCD” do PM. Por meio de valores de propriedades (valores rotulados), as regras de transformação substituem as operações definidas no perfil *swxRTOS* pelas operações correspondentes definidas no PM.

Durante a transformação, uma classe chamada “X” é criada no PSM com base nos valores da propriedade *rtService*. Por sua vez, a operação “sendPutMsg” da classe “CtrlMsg” é substituída pela operação nominada “Put”, com base na propriedade *opTarget*. Uma associação é definida entre a classe “CtrlMsg” (já existente no PIM) e a nova classe “X”. Da mesma forma, uma classe chamada “CDDX_LCD” é criada no PSM, e a operação “wriStrLCD” é substituída pela operação “writeStreAt55”. O resultado da transformação modelo é a geração da PSM para a plataforma selecionada.

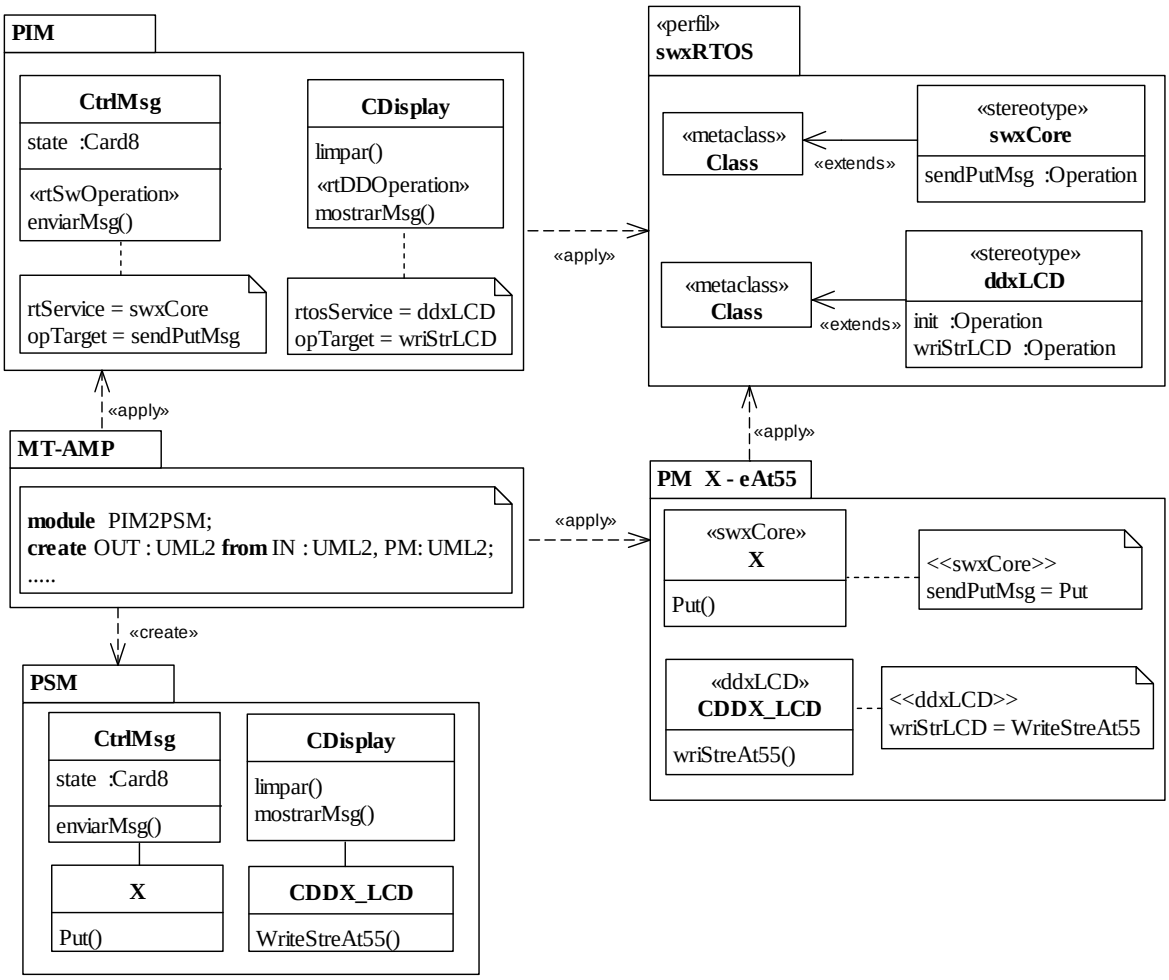

Figura 36 - Exemplo de aplicação da transformação MT-AMP

4.9 TRANSFORMAÇÃO DE MODELOS MT-PROAPES

Esta seção apresenta a transformação de modelos PIM-comportamental em PSM-comportamental, denominada MT-PROAPES (*Model Transformation based on Profile for Modeling Application and Platform of Embedded Software*), baseada no método PI-MT e proposta nesta pesquisa. No contexto desta tese, um modelo comportamental representa o comportamento dinâmico de um sistema de *software* e, desse modo, a transformação MT-PROAPES suporta o uso de modelos na forma de diagramas de sequência da UML. Dessa forma, a MT-PROAPES tem como foco a modelagem dinâmica (comportamental) de um sistema de *software*.

Assim como acontece na transformação MT-AMP, um PM explicitamente definido é utilizado como entrada da transformação MT-PROAPES e realiza um refinamento de modelos UML. Nesta transformação, os serviços de plataforma definidos de forma abstrata no PIM, com base no perfil *swxRTOS*, são substituídos pelos serviços específicos de plataforma, definidos no PM.

O perfil de ligação utilizado pela transformação MT-PROAPES é o *dynRTOS*, integrante do perfil PROAPES (SOARES, 2012). Desse modo, a transformação é realizada com base em informações armazenadas nas propriedades (*tagged-values*) relacionadas com os estereótipos definidos no perfil *dynRTOS*.

A visão geral da transformação de modelos MT-PROAPES é apresentada na Figura 37. Os retângulos coloridos, ilustrados nessa figura, representam o fluxo da transformação por meio dos modelos de entrada e saída (PM, PIM e PSM), e dos modelos de transformação (MT-PROAPES.atl e UML2Copy.atl). Os demais retângulos representam os metamodelos padrão utilizados na definição desses modelos (metamodelo UML, metamodelo ATL, MOF, perfil *swxRTOS* e perfil *dynRTOS*).

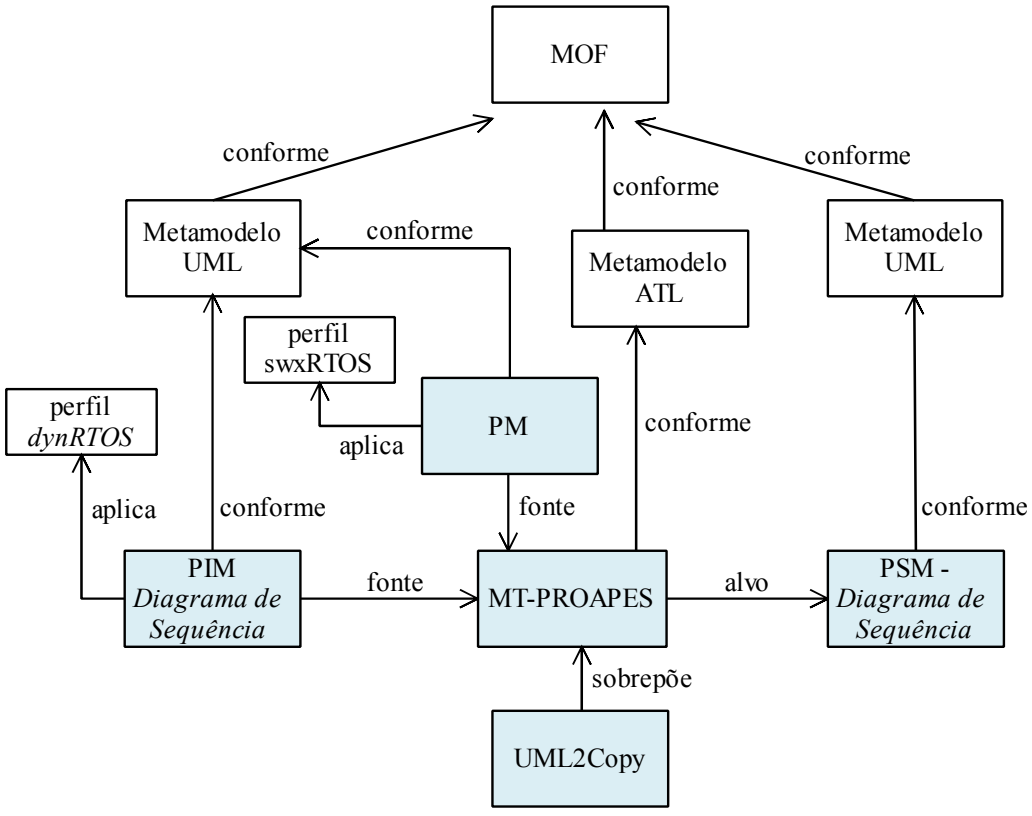

Figura 37 – Padrão da Transformação MT-PROAPES

A transformação MT-PROAPES é realizada por dois módulos ATL: i) UML2Copy.atl; e ii) MT-PROAPES.atl. Do mesmo modo que acontece na transformação MT-AMP, o módulo UML2Copy.atl, proposto por Wagellar *et al*. (2010) e publicamente disponibilizado[7], realiza a cópia de todos os elementos de um modelo UML. Por sua vez, o módulo MT-PROAPES contém as regras específicas de refinamento, responsáveis por incorporar ao modelo PSM detalhes específicos da plataforma alvo. Desta forma, por meio da técnica de Sobreposição de Módulos, apresentada no Capítulo 3, as regras do módulo UML2Copy.atl são reutilizadas e sobrepostas quando necessário pelas regras do módulo MT-PROAPES.atl.

4.9.1 Perfil de Ligação *dynRTOS*

O perfil de ligação *dynRTOS*, proposto por Soares (2012), representa os conceitos de modelagem dinâmicos para o desenvolvimento dirigido a modelos de *software* embarcado baseado no RTOS *X Real-Time Kernel*. Este perfil faz parte do Perfil para a Modelagem da Aplicação e Plataforma de *Software* Embarcado (PROAPES - *Profile for Modeling Application and Platform of Embedded Software*).

A Figura 38 ilustra a arquitetura do perfil de ligação *dynRTOS*, composto pelos subperfis *dynSwRTOS* e *dynDDRTOS*. O subperfil *dynSwRTOS* visa modelar o comportamento dinâmico de sistemas de *software* de execução concorrente, por sua vez, o subperfil *dynDDRTOS* visa modelar o comportamento dinâmico de *drivers* de dispositivos. Estes subperfis definem os estereótipos a serem aplicados no modelo PIM para marcar as mensagens relacionadas com serviços de *software* ou de *hardware* relacionados a uma plataforma específica.

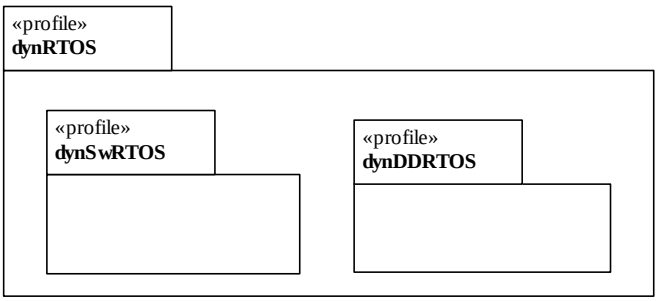

Figura 38 – Arquitetura geral do perfil *dynRTOS*

[7] http://ssel.vub.ac.be/ssel/research:mdd:casestudies

A Tabela 6 apresenta os estereótipos definidos pelo subperfil *dynSwRTOS* e que representam serviços de *software* de RTOS. O estereótipo *dynActionSwRTOS* representa conceitos de ações gerais especificadas em mensagens de diagramas de sequência. O *dynActionSwRTOS* define as seguintes propriedades (*tagged-values*):

- *opSwTarget*: indica o serviço de *software* de RTOS utilizado, com base em uma plataforma abstrata.
- *paSwTarget*: indica as propriedades definidas no estereótipo do perfil *swxRTOS* que correspondem aos parâmetros dos serviços de RTOS.
- *rtSwService*: indica o nome da classe do perfil de plataforma responsável por definir o serviço de RTOS em um PM (voltado a uma plataforma específica) que será vinculado ao PIM.

Todos os outros estereótipos definidos no subperfil *dynSwRTOS,* descritos na Tabela 6, são especializações do estereótipo *dynActionSwRTOS*. Cada um desses estereótipos define propriedades relacionadas com os parâmetros do serviço de RTOS relacionado. A Figura 39 ilustra o subperfil *dynSwRTOS*.

Tabela 6 – Estereótipos do perfil *dynSwRTOS*

Estereótipo	Descrição de ação do estereótipo
dynActionSwRTOS	Ações gerais de *software* em projetos baseados em RTOS
dynCheckMessage	Detecção de mensagem na caixa de entrada de uma *thread*
dynInitSched	Ativação do escalonador
dynSendMsg	Envio de mensagem de uma *thread* para outra
dynReceiveMsg	Recebimento de mensagem por uma *thread*
dynActivateThread	Criação de *threads*
dynInitRTOS	Inicialização de estruturas internas do RTOS
dynSleepThreadTime	Interrupção de uma *thread* por determinado período de tempo
dynGetIdThread	Obter identificação de uma *thread* do RTOS

O subperfil *dynDDRTOS*, ilustrado na Figura 40, define os estereótipos responsáveis por marcar as mensagens do PIM relacionadas com serviços de *drivers* de dispositivos do RTOS. Os estereótipos definidos por este subperfil são apresentados na Tabela 7.

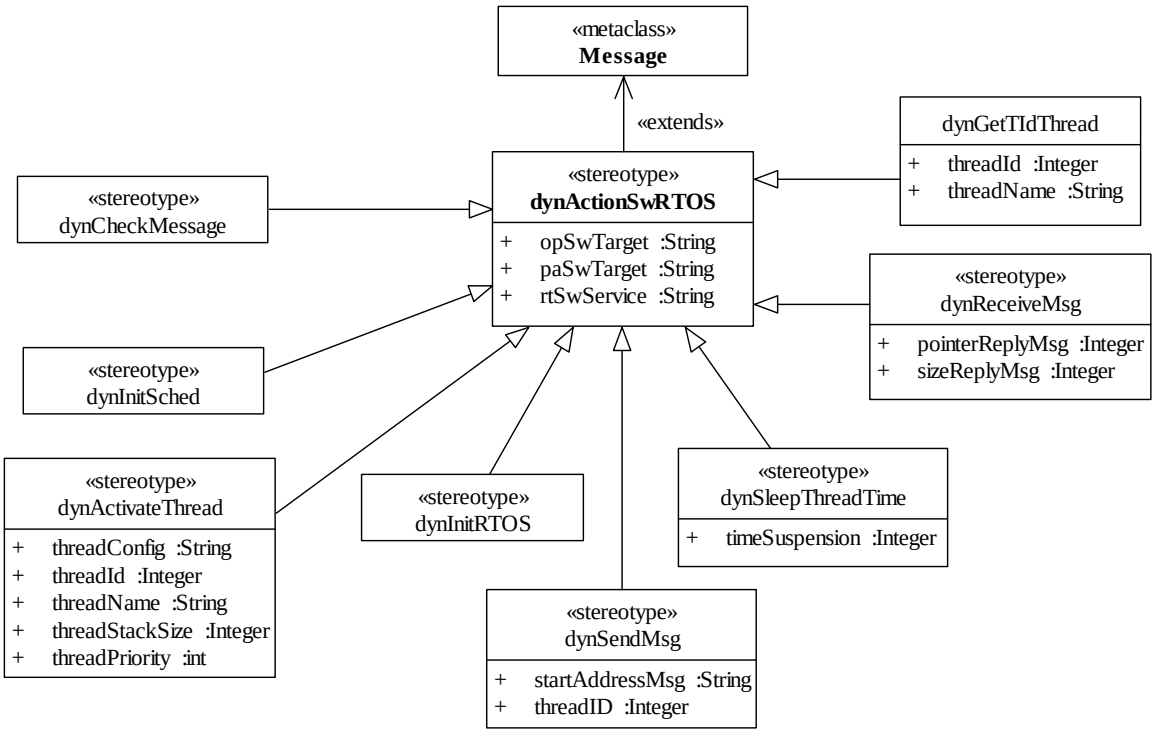

Figura 39 – Subperfil *dynSwRTOS*
Fonte: Soares (2012)

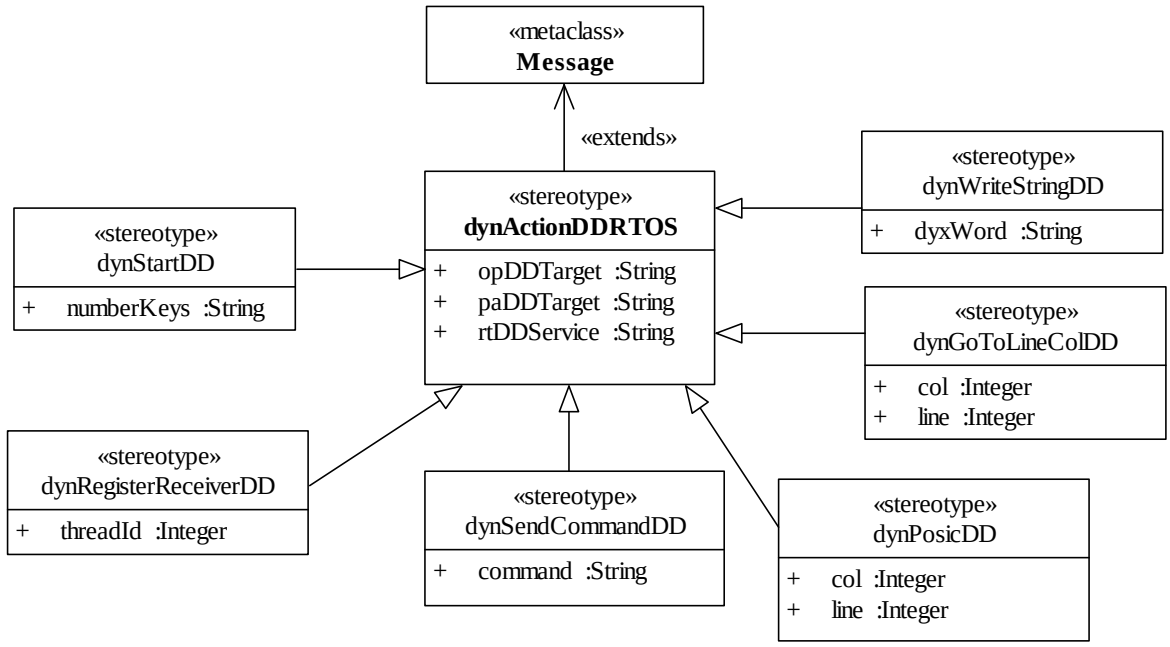

Figura 40 – Subperfil *dynDDRTOS*
Fonte: Soares (2012)

Tabela 7 – Estereótipos definidos no subperfil *dynDDRTOS*

Estereótipo	Descrição
dynActionDDRTOS	Ações gerais de *drivers* de dispositivo em projetos baseados em RTOS
dynStartDD	Inicialização do *driver* de dispositivo
dynRegisterReceiverDD	Ações de registro de recebimento do identificador da *thread* de controle
dynSendCommandDD	Ações de envio de execução de comando para um *driver* de dispositivo
dynGoToLineColDD	Configura o cursor para determinada linha e coluna
dynWriteStringDD	Escreve um texto no visor de LCD
dynPosicDD	Ações de posicionamento de um comando para um *device driver*

O estereótipo *dynActionDDRTOS* representa conceitos de ações gerais relacionadas com dispositivos de *hardware* em diagramas de sequência e define as seguintes propriedades:

- *opDDTarget*: indica o serviço de *driver* de dispositivo utilizado, com base em uma plataforma abstrata.
- *paDDTarget*: indica as propriedades definidas no estereótipo do perfil *swxRTOS* que correspondem aos parâmetros do serviço de RTOS relacionado.
- *rtDDService*: indica o nome da classe do perfil de plataforma responsável por marcar o serviço de *driver* de dispositivo em um PM (voltado a uma plataforma específica) que será vinculado ao PIM.

Os outros estereótipos, apresentados na Tabela 7, generalizam as propriedades do estereótipo *dynActionDDRTOS* e, por sua vez, definem propriedades específicas referentes aos parâmetros do respectivo serviço de RTOS utilizado.

O módulo MT-PROAPES.atl, principal módulo da transformação do PIM-comportamental em PSM-comportamental, é apresentado na próxima subseção.

4.9.2 Implementação da Transformação MT-PROAPES

A transformação MT-PROAPES foi realizada no ambiente TopCased baseada no Eclipse. A configuração da técnica de sobreposição é ilustrada nas Figuras 41 e 42. A Figura 41 apresenta a configuração da transformação em si, dirigida pelo módulo UML2Copy.atl. Os modelos de entrada (PIM.uml e PM.uml) e o modelo de saída (PSM.uml) são definidos.

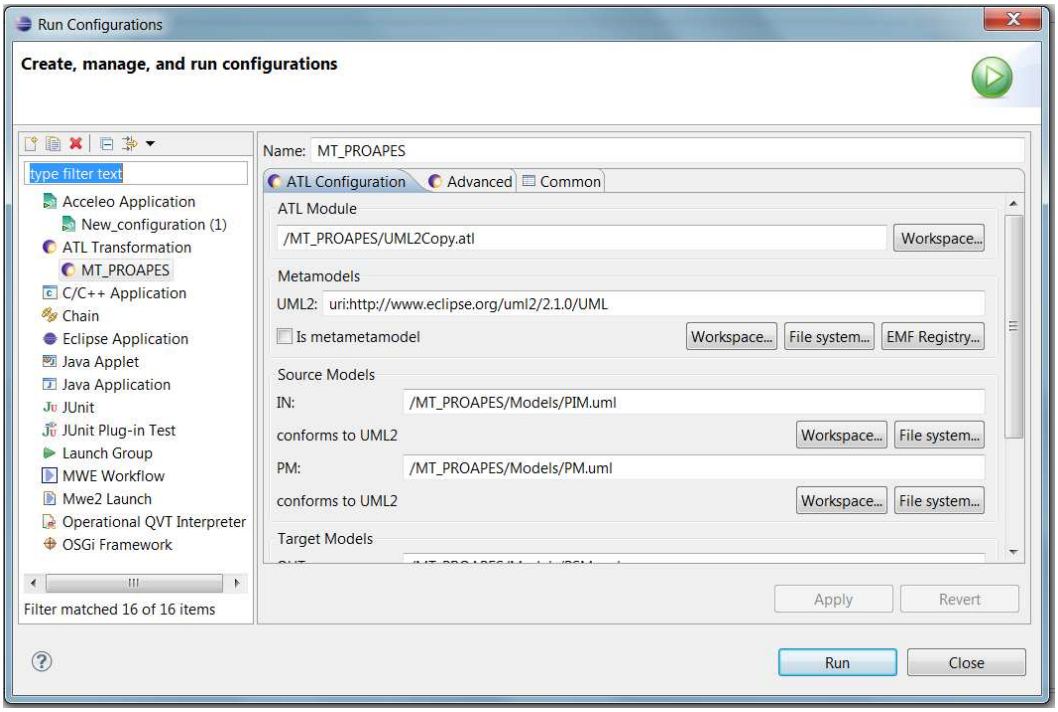

Figura 41 - Tela de configuração da Transformação MT-PROAPES

Por sua vez, a Figura 42 ilustra a configuração da sobreposição do módulo MT-PROAPES.atl sobre o módulo UML2Copy.atl. Quando o módulo MT-PROAPES.atl é sobreposto ao módulo UML2Copy.atl, o primeiro é adicionado à transformação base.

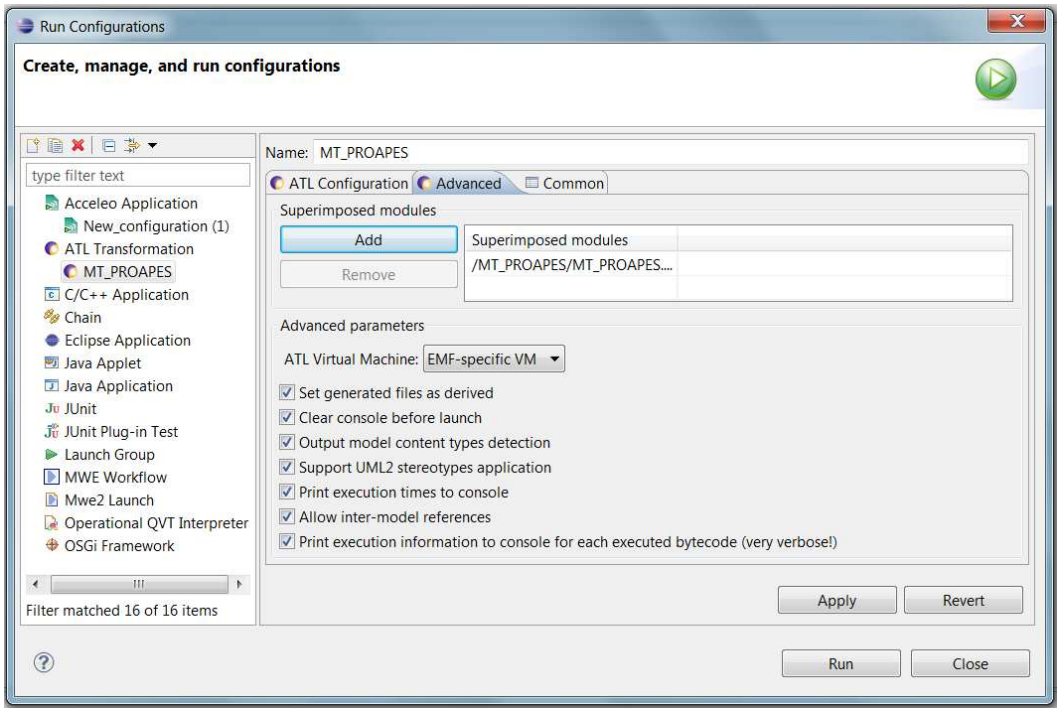

Figura 42 - Tela de configuração da Sobreposição do Módulo MT-PROAPES

O módulo ATL proposto contém uma seção *header*, *helpers* e regras de transformação que serão detalhados nas próximas subseções.

4.9.2.1 Seção Header

A seção *header* (Figura 43) define o nome do referido módulo (MT-PROAPES.atl) e declara os modelos de entrada e o modelo alvo da transformação, sendo eles: modelo IN - refere-se ao modelo-fonte (PIM); modelo PM - refere-se ao modelo de plataforma (PM); e, finalmente, modelo OUT - refere-se ao PSM e é criado como resultado da transformação. Os modelos PIM, PM e PSM estão em conformidade com o metamodelo UML2, neste caso, especificado no metamodelo Eclipse UML2.

```
module MT_PROAPES; --superimposed on UML2Copy
create OUT : UML2 from IN : UML2, PM: UML2;
```

Figura 43 – Cabeçalho do módulo MT-PROAPES.atl

4.9.2.2 Helpers do Módulo MT-PROAPES

O módulo MT-PROAPES.atl possui diversos *helpers* que definem operações no contexto de um elemento do modelo UML, como uma mensagem. A Tabela 8 apresenta os *helpers* definidos no módulo MT-PROAPES.atl. Alguns destes *helpers* serão apresentados a seguir.

O *helper* “getTagVal”, como exemplo, recupera o valor de uma propriedade (*tagged value* - valor rotulado) associada com um estereótipo aplicado a um elemento do modelo. Para tal, recebe o nome de um estereótipo e da propriedade correspondente. A Figura 44 apresenta, como exemplo, o *helper* “getPMClass” que realiza uma busca no modelo de plataforma pela classe anotada com o estereótipo recebido como parâmetro. Finalmente, outro exemplo é apresentado na Figura 45 que ilustra o *helper* “isdynSendMsg” responsável por verificar se um elemento do modelo, neste caso uma *mensagem*, possui o estereótipo *dynSendMsg* aplicado.

```
helper def : getPMClass (sName : String) : UML2!Class =
    UML2!Class.allInstancesFrom('swxRTOS')->select(c | c.getAppliedStereotypes()->exists
    (st|st.name = sName)).first();
```

Figura 44 – *Helper* “*getPMClass*”

```
helper context UML2!"uml::Message" def : isdynSendMsg() : Boolean =
        self.getAppliedStereotypes()->exists(st|st.name = 'dynSendMsg');
```

Figura 45 – *Helper* “*isdynSendMsg*”

Tabela 8 – *Helpers* do MT-PROAPES.atl

Nome do *helper*	**Descrição do *helper***
getTagVal	Retorna o valor de uma propriedade (*tagged value*) associada a um estereótipo específico. Recebe como parâmetros o nome da propriedade e o nome do estereótipo.
getPMClass	Busca no PM por uma classe que possua um dado estereótipo aplicado. Recebe como parâmetro o nome do estereótipo pertencente ao subperfil *swxRTOS* a ser localizado em uma classe do PM.
isStereotype	Verifica se uma *Mensagem* do modelo-fonte possui um estereótipo aplicado.
isdynSendMsg	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynSendMsg* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynCheckMessage	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynCheckMessage* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynInitSched	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynInitSched* aplicado. Recebe o nome deste estereótipo com parâmetro.
isdynReceiveMsg	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynReceiveMsg* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynActiveThread	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynActiveThread* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynInitRTOS	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynInitRTOS* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynSleepThreadTime	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynSleepThreadTime* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynGetIdThread	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynGetIdThread* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynStartDD	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynStartDD* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynRegisterReceiverDD	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynRegisterReceiverDD* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynSendCommandDD	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynSendCommandDD* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynGoToLineColDD	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynGoToLineColDD* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynWriteStringDisplayDD	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynWriteStringDisplayDD* aplicado. Recebe o nome deste estereótipo como parâmetro.
isdynPosicDD	Verifica se uma *Mensagem* do modelo-fonte possui o estereótipo *dynPosicDD* aplicado. Recebe o nome deste estereótipo como parâmetro.

4.9.2.3 Regras do Módulo MT-PROAPES

O módulo MT-PROAPES.atl implementa regras específicas para cada tipo de estereótipo aplicado a uma mensagem do modelo PIM. A Tabela 9 apresenta essas regras e suas respectivas definições.

Tabela 9 – Regras do módulo MT-PROAPES

Nome da *regra*	**Descrição da *regra***
MessagedynRTOS	Realiza a cópia dos atributos de uma mensagem independentes de plataforma.
MessagedynCheckMessage	Incorpora características específicas de plataforma em mensagens que verificam a existência de uma notificação.
MessagedynInitSched	Incorpora características específicas de plataforma em mensagens que realizam a ativação de um *scheduler*.
MessagedynSendMessage	Incorpora características específicas de plataforma em mensagens que enviam uma notificação de ação de uma *thread* para outra.
MessagedynReceiveMessage	Incorpora características específicas de plataforma em mensagens que recebem de uma *thread* a notificação de uma ação a ser realizada.
MessagedynActivateThread	Incorpora características específicas de plataforma em mensagens que realizam a criação de uma *thread*.
MessagedynInitRTOS	Incorpora características específicas de plataforma em mensagens que realizam a inicialização de estruturas internas do RTOS.
MessagedynSleepThreadTime	Incorpora características específicas de plataforma em mensagens que realizam a interrupção de uma *thread* por um tempo definido.
MessagedynGetIdThread	Incorpora características específicas de plataforma em mensagens que recuperam a identificação de uma *thread* em execução.
MessagedynStartDD	Incorpora características específicas de plataforma em mensagens que realizam a inicialização do *device driver*.
MessagedynRegisterReceiverDD	Incorpora características específicas de plataforma em mensagens que recebem o registro do identificador da *thread* de controle do *device driver*.
MessagedynSendCommandDD	Incorpora características específicas de plataforma em mensagens que enviam um comando de execução para um *device driver*.
MessagedynGoToLineColDD	Incorpora características específicas de plataforma em mensagens que posicionam o cursor em uma linha e uma coluna específica do *device driver*.
MessagedynWriteStringDD	Incorpora características específicas de plataforma em mensagens que utilizam operações que escrevem uma *string* em um *display* LCD.
MessagedynPosicDD	Incorpora características específicas de plataforma em mensagens que representam ações de posicionamento de um comando para um *device driver*.

O mecanismo de herança oferecido pela linguagem ATL é utilizado na transformação MT-PROAPES, possibilitando que uma regra herde as funcionalidades de outra regra. O objetivo é permitir que propriedades comuns a diversas regras sejam agrupadas em uma regra base, ou super-regra. Desta forma, a partir de uma super-regra, outras podem ser especificadas. Cada regra derivada ou sub-regra contempla as propriedades da super-regra e acrescenta a ela propriedades específicas. Desta forma, a herança acarreta melhoria da reusabilidade e da manutebilidade do código desenvolvido (JOUAULT e KURTEV, 2006).

O conceito de herança é oferecido em ATL por meio de regras abstratas (*abstract rules*). Desta forma, é possível definir uma super-regra abstrata e diversas sub-regras que estendem esta super-regra. A regra *MessagedynRTOS* é uma super-regra que manipula os

atributos gerais de uma mensagem estereotipada, independente do estereótipo aplicado a esta. As demais regras *MessagedynCheckMessage*, *MessagedynInitSched*, *MessagedynSendMessage*, *MessagedynReceiveMessage*, *MessagedynActivateThread*, *MessagedynInitRTOS*, *MessagedynSleepThreadTime*, *MessagedynGetIdThread*, *MessagedynStartDD*, *MessagedynRegisterReceiverDD*, *MessagedynSendCommandDD*, *MessagedynGoToLineColDD*, *MessagedynWriteStringDD* e *MessagedynPosicDD* estendem a regra *MessagedynRTOS* e incluem propriedades específicas que realizam as alterações dependentes de plataforma, com base no estereótipo aplicado à *mensagem*.

A Figura 46 ilustra a super-regra *MessagedynRTOS* que opera no contexto de mensagens estereotipadas pertencentes ao modelo-fonte. Esta regra realiza a cópia dos atributos de uma mensagem que são independentes de plataforma, ou seja, não sofrem alterações devido à plataforma de RTOS adotada. A regra *MessagedynRTOS* é uma regra abstrata, estendida por outras regras do módulo MT_PROAPES.atl que adicionam a esta novas propriedades.

Na regra *MessagedynRTOS* todas as instâncias da metaclasse *Message* do metamodelo UML2 existentes no modelo PIM disparam a execução dessa regra. A linguagem ATL utiliza o conector '<-' para especificar uma atribuição. Assim, a parte “from” dessa regra seleciona no modelo-fonte apenas os elementos UML do tipo “Message” (indicados por “s”). Na parte “from” de uma regra ATL também é possível adicionar restrições. A regra “*MessagedynRTOS*” considera apenas as operações do modelo-fonte anotadas com estereótipos, por meio da seguinte definição de restrição “s.isStereotype()”.

```
abstract rule MessagedynRTOS {
    from s : UML2!"uml::Message" (
    if thisModule.inElements->includes(s) and s.isStereotype()
    then
        s->oclIsTypeOf(UML2!"uml::Message")
    else false endif)
    to t : UML2!"uml::Message" mapsTo s (
        visibility <- s.visibility,
        messageSort <- s.messageSort,
        eAnnotations <- s.eAnnotations,
        ownedComment <- s.ownedComment,
        clientDependency <- s.clientDependency,
        nameExpression <- s.nameExpression,
        receiveEvent <- s.receiveEvent,
        sendEvent <- s.sendEvent,
        connector <- s.connector,
        argument <- s.argument)
}
```

Figura 46 - Regra *MessagedynRTOS*

A parte “to” da regra “*MessagedynRTOS*”, ilustrada na Figura 46, cria elementos do tipo UML2!"uml::Message" no modelo PSM. Considerando a regra “*MessagedynRTOS*”, a cópia da operação será realizada com base nos valores das propriedades (visibilidade, conector, dentre outras) especificados no elemento do modelo original. Todas estas propriedades estão definidas na metaclasse *Message* do metamodelo UML (OMG, 2011a).

A Figura 47 ilustra a regra *MessagedynSendMsg* válida somente no contexto de mensagens que possuem o estereótipo *dynSendMsg* aplicado. O estereótipo *dynSendMsg* é definido no modelo de domínio do *dynSwRTOS* e, portanto, aplicado em mensagens que operam no contexto de *software*. Esta regra estende a regra *MessagedynRTOS* e opera em elementos do PIM responsáveis por enviar mensagens para *threads* do RTOS em execução.

```
rule MessagedynSendMsg extends MessagedynRTOS {
    from s : UML2!"uml::Message"(
    if thisModule.inElements->includes(s) and s.isdynSendMsg()
    then
        s->oclIsTypeOf(UML2!"uml::Message")
    else false endif)
    to t : UML2!"uml::Message" mapsTo s (
        name <- thisModule.getswxRTOSClass(s.getTagVal('dynSendMsg',
        'rtSwService')).getTagVal(s.getTagVal('dynSendMsg',
        'rtSwService'),s.getTagVal('dynSendMsg', 'opSwTarget')).name +
        '(' + s.getTagVal('dynSendMsg', 'nameThreadRec') + ',' +
        s.getTagVal('dynSendMsg', 'nameThreadSend') + ')')
}
```

Figura 47 – Regra *MessagedynSendMsg*

A regra *MessagedynRTOS* pode ser explicada por meio de quatro passos principais:

1. O *helper getPMClass* é chamado para realizar a busca no modelo de plataforma por uma classe anotada com um estereótipo do perfil *swxRTOS* cujo nome seja igual ao conteúdo da propriedade chamada *rtSwService*. A propriedade (*tagged-value) rtSwService* está definida no estereótipo *dynActionSwRTOS*, lembrando que *dynSendMsg* é uma especialização de *dynActionSwRTOS* (conforme descrito na Subseção 4.8.1);
2. Em seguida, o *helper getTagVal* é chamado para recuperar o valor da propriedade *opSwTarget* correspondente a uma operação (serviço) do RTOS. A propriedade *opSwTarget* está definida no estereótipo *dynActionSwRTOS*, lembrando que *dynSendMsg* é uma especialização de *dynActionSwRTOS* (conforme descrito na Subseção 4.8.1);

3. A mensagem criada no modelo PSM recebe, então, o nome da respectiva operação localizada no PM (Passo 2);
4. Os parâmetros da operação (criada no passo 3) são definidos por meio das propriedades *StartAddressMsg* e *threadID* relacionadas ao estereótipo *dynSendMsg* e recuperados pelo *helper getTagVal*.

A Figura 48 ilustra a regra *MessagedynRegisterReceiverDD* definida de forma similar à regra *MessagedynSendMessage* (Figura 47). Entretanto, a regra *MessagedynRegisterReceiverDD* é executada para mensagens que possuem o estereótipo *dynStartDD* aplicado. Esse estereótipo é definido no modelo de domínio do *dynDDRTOS* e, portanto, aplicado em mensagens que operam no contexto de *drivers* de dispositivos. Assim, a principal diferença na definição da regra *MessagedynSendMessage* e *MessagedynStartDD* está no nome das propriedades relacionadas a estes estereótipos.

```
rule MessagedynRegisterReceiverDD extends MessagedynRTOS {
    from s : UML2!"uml::Message" (
    if thisModule.inElements->includes(s) and s.isdynRegisterReceiverDD()
    then
        s->oclIsTypeOf(UML2!"uml::Message")
    else false endif)
    to t : UML2!"uml::Message" mapsTo s (
        name <- thisModule.getPMClass(s.getTagVal('dynRegisterReceiverDD',
        'rtDDService')).getTagVal(s.getTagVal('dynRegisterReceiverDD',
        'rtDDService'),s.getTagVal('dynRegisterReceiverDD', 'opDDTarget')).name +
        '(' + s.getTagVal('dynRegisterReceiverDD', 'threadID') + ')')
}
```

Figura 48 – Regra *MessagedynRegisterReceiverDD*

A regra *MessagedynRegisterReceiverDD* realiza os seguintes passos:

1. O *helper getPMClass* é chamado para realizar a busca no modelo de plataforma por uma classe anotada com um estereótipo do perfil *swxRTOS* cujo nome seja igual ao conteúdo da *tagged-value* chamada *rtDDService*. A propriedade *rtSwService* está definida no estereótipo *dynActionDDRTOS*, lembrando que *dynRegisterReceiver* é uma especialização de *dynActionSwRTOS*;
2. Em seguida, o *helper getTagVal* é chamado para recuperar o valor da propriedade *opDDTarget* correspondente a uma operação (serviço) do RTOS. A *tagged-value opSwTarget* está definida no estereótipo *dynActionDDRTOS*, lembrando que *dynRegisterReceiver* é uma especialização de *dynActionDDRTOS*;

3. A operação localizada no PM é definida, então, como nome da mensagem que está sendo manipulada pela respectiva regra;
4. Os parâmetros da operação, definidos no PIM por meio da propriedade *threadID* relacionada ao estereótipo *dynRegisterReceiver*, são recuperados pelo *helper getTagVal* e atribuídos ao corpo da mensagem.

As outras regras mencionadas operam de forma similar, executando os ajustes necessários para tratar de um domínio específico, relacionado com um determinado estereótipo pertencente ao perfil *dynRTOS*.

4.9.3 Exemplo de Aplicação da Transformação MT-PROAPES

A Figura 49 traz um exemplo de aplicação da transformação MT-PROAPES. Esse exemplo faz parte do caso de estudo realizado nesta tese e que implementa um sistema Simulador de Alarme (SA), detalhado no Capítulo 5.

O modelo PIM, ilustrado na Figura 49, representa um diagrama de sequência, denominado “Mostrar Palavra”. Esse diagrama é integrante do módulo Gerenciar *Display* do SA e define as funcionalidades de gerenciamento do *display*.

O PIM é representado pelo diagrama de sequência “Mostrar Palavra” apresenta a transformação de modelos MT-PROAPES realizada para duas plataformas distintas, sendo elas: RTOS X *Real-Time Kernel* em placas eAt55 (X – eAt55) e RTOS X *Real-Time Kernel* em placas eLPC48 (X – eLPC48).

O modelo PIM possui duas mensagens: *PosicionarCursorDisplay* (anotada pelo estereótipo <<dynPosicDD>>) e *EscreverStringDisplay* (anotada pelo estereótipo <<dynWriteStringDD>>). O estereótipo <<dynPosicDD>> denota um serviço de RTOS que realiza ações de posicionamento de um comando para um *display*, por sua vez, o estereótipo <<dynWriteStringDD>> aponta um serviço de RTOS que escreve um texto no visor de LCD.

As principais regras que compõe o módulo MT-PROAPES e utilizadas para executar a transformação exemplificada na Figura 49 são: *MessagedynPosicDD* e *MessagedynWriteStringDD*. A regra *MessagedynPosicDD* incorpora características específicas de plataforma em mensagens que representam ações de posicionamento de um comando para um *device driver* e considera os valores atribuídos às *tagged-values* relacionadas com o estereótipo <<dynPosicDD>> (*rtDDService*, *opDDTarget*, *linLCD* e

colLCD). A *tagged-value rtDDService* auxilia na busca da respectiva classe do PM que contém o serviço de RTOS utilizado, por sua vez, a *tagged-value opDDTarget* armazena o nome de serviço de plataforma definido no perfil de plataforma. As *tagged-values linLCD* e *colLCD* guardam os valores dos parâmetros necessários para a realização do serviço.

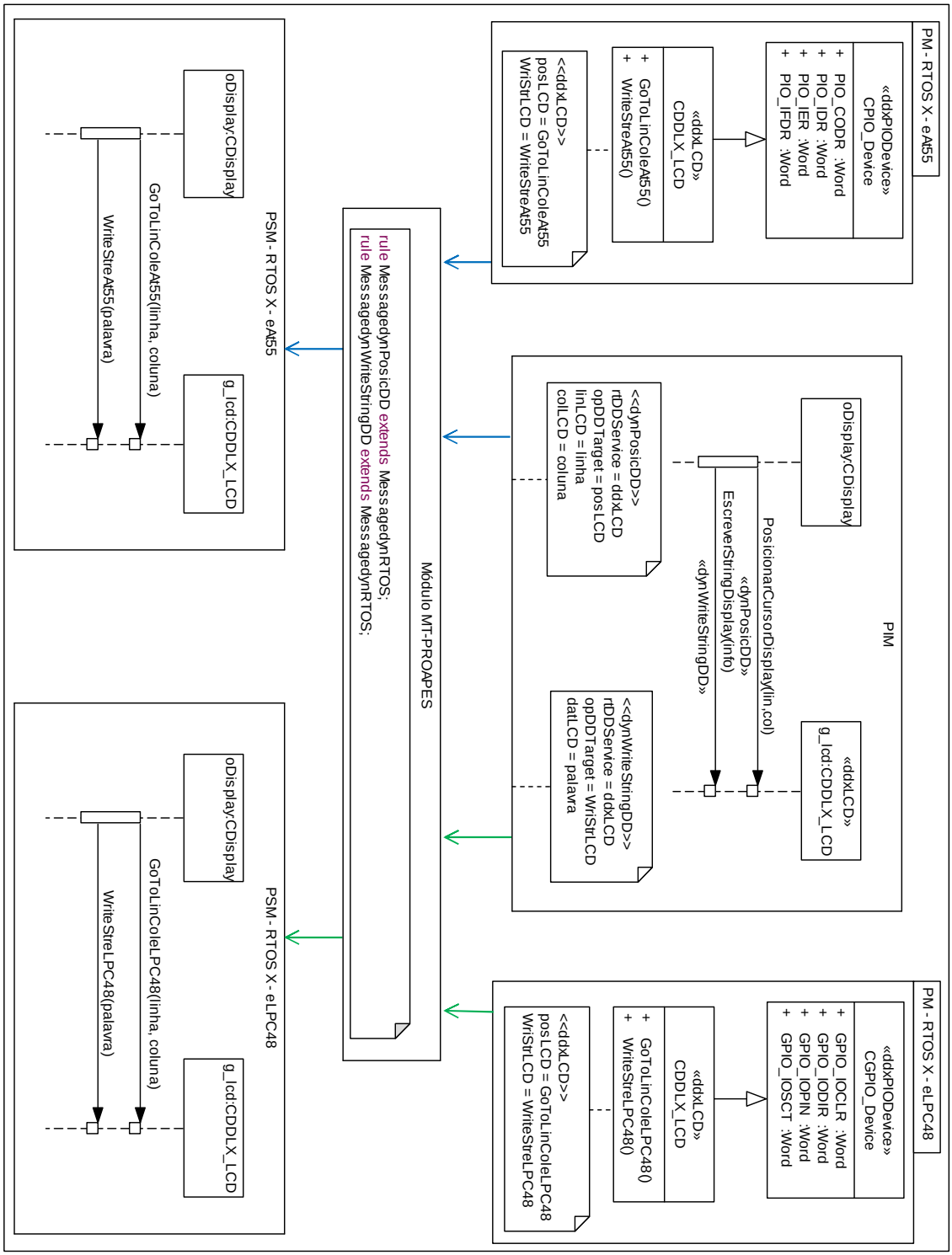

Figura 49 – Exemplo de Aplicação da Transformação MT-PROAPES

A regra *MessagedynWriteStringDD* incorpora características específicas de plataforma em mensagens que utilizam operações que escrevem uma *string* em um display LCD e recupera os valores atribuídos às *tagged-values* relacionadas com o estereótipo <<dynWriteStringDD>> (*rtDDService*, *opDDTarget*, *datLCD*). A *tagged-value datLCD* armazena o valor do parâmetro relacionado com o serviço de RTOS utilizado, sendo que as tagged-values *rtDDService*, *opDDTarget* possuem a mesma função apresentada na descrição da regra *MessagedynWriteStringDD*.

Para melhor exemplificar, considerando a mensagem *PosicionarCursorDisplay*, definida no PIM e anotada com o estereótipo <<dynPosicDD>>. No PSM X – eAt55 gerado, esta mensagem é transformada no serviço *GoToLinColeAt55* com base nos seguintes passos realizados pela regra *MessagedynPosicDD* definida no módulo MT-PROAPES:

- Primeiramente, o valor da tagged-*value rtDDservice* = *ddxLCD* é recuperado. Em seguida, uma busca é realizada no respectivo modelo de plataforma (PM X – eAt55) visando localizar a classe existente nesse PM marcada pelo estereótipo *ddxLCD*;
- Depois de localizar a respectiva classe no PM, nominada CDDX_LCD, é preciso recuperar o valor da *tagged-value posLCD* (associada ao estereótipo <<ddxLCD>>). Antes disso, porém, é preciso recuperar o valor da propriedade *opDDTarget* (associada ao estereótipo <<ddxPosicDD>> e que contém o nome do serviço a ser buscado no PM (*posLCD*).

Ambos os PSMs gerados para as plataformas X – eAt55 e X –eLPC48 são ilustrados na Figura 49. A tradução da mensagem “EscreverStringDisplay” definida no PIM acontece de forma similar à tradução da mensagem “PosicionarCursorDisplay”.

4.10 VERIFICAÇÃO E TESTES

No ciclo de desenvolvimento de *software*, uma vez que uma transformação de modelos é especificada e implementada em uma linguagem de transformação, o processo de verificação visa determinar se a implementação dessa transformação satisfaz suas especificações (LAMARI, 2007).

As transformações de modelos são definidas por meio de programas ou módulos de transformação e, portanto, consideradas como *software*. Desse modo, técnicas bem

estabelecidas de Engenharia de *Software* podem ser utilizadas para testar e verificar essas transformações de modelos (KÜSTER, 2004; MCQUILLAN e POWER, 2009; LAMARI, 2007).

Em adição, o desenvolvimento de transformações de modelos não é tarefa trivial e é propenso a erros (KÜSTER, 2004). A criação de uma transformação de modelos não é completa sem a realização de testes e a verificação de que a implementação satisfaz suas especificações. A verificação aumenta a confiabilidade e a usabilidade de transformações do modelo. Quando uma transformação não funciona corretamente pode inserir erros no *software* gerado. Desta forma, é necessário testar e verificar a transformação de modelos a fim de detectar erros o quanto antes.

Teste de *software* é amplamente utilizado para realizar verificação e testes de transformações de modelos são os (MCQUILLAN e POWER, 2009; KÜSTER e ABD-EL-RAZIK, 2006; LAMARI, 2007; WANG *et al*., 2008). Esta abordagem consiste em verificar se uma transformação funciona para um determinado conjunto de casos de testes composto de alguns modelos de entrada, sem tentar validá-lo para todo o espaço de entrada. Embora não seja possível provar completamente a corretude da transformação por meio de teste de *software*, essa abordagem é muito útil para identificar erros de uma forma viável e possibilita verificar as transformações desenvolvidas sem precisar abstrair qualquer uma das propriedades estruturais ou comportamentais das transformações (GOGOLLA e VALLECILLO, 2011).

Tradicionalmente, o teste destina-se a executar um programa em situações representativas previamente selecionadas por meio da definição de casos de teste adequados. O teste de *software* é uma parte importante do processo de desenvolvimento de *software*. Utilizado para revelar defeitos em um sistema, para assegurar que o sistema está em conformidade com sua especificação e para verificar que o sistema se comporta da maneira pretendida. A atividade de teste precisa definir um conjunto de casos de teste com alta probabilidade de encontrar um defeito não revelado.

O teste realizado nesta pesquisa foi baseado na técnica funcional, também conhecido como teste de "*caixa-preta*", que têm como foco os requisitos funcionais do *software* testado. Em testes funcionais a geração dos casos de teste é independente da implementação da transformação de modelos. Testes de "*caixa-preta*" são amplamente utilizados para a verificação de transformações de modelo no contexto da MDE/MDA (GUERRA, 2012; FIORENTINI *et al*., 2010; SEN *et al*., 2009; FLEUREY *et al*., 2009; LAMARI, 2007; WANG *et al*., 2008).

No esforço de realizar os testes funcionais, os seguintes passos foram identificados e realizados:

1. Casos de teste foram gerados com base na especificação das transformações de modelos e dos critérios de teste especificados. No conjunto de casos de teste gerado, cada um dos casos de teste define um modelo de entrada e os resultados esperados, ou seja, o modelo alvo esperado como resultado da execução da transformação de modelos.
2. Os testes foram executados, segundo o conjunto de casos de teste selecionado.
3. Os resultados obtidos na execução foram avaliados, ou seja, o resultado esperado para cada caso de teste foi comparado com o resultado obtido.
4. Alterações dos programas (módulos) de implementação foram realizadas com base nos erros encontrados durante a realização do Passo 3.

4.10.1 Geração dos Casos de Teste

A definição do conjunto de casos de teste requer o conhecimento das características e valores que podem ser assumidos por um modelo e que podem ter influência sobre o comportamento da transformação. No âmbito desta tese, esses dados são caracterizados por duas dimensões: i) a estrutura definida pelas relações especificadas em uma transformação de modelos, e ii) os valores atribuídos às propriedades dos elementos de um modelo. Além disso, os modelos de teste são escolhidos para satisfazer requisitos (previamente definidos) e devem aumentam a capacidade de detectar erros na transformação de modelos (SEN *et al*., 2008).

O método de “particionamento por categoria”, proposto por Ostrand e Balcer (1988), foi utilizado para a criação do conjunto de casos de teste nesta pesquisa. De acordo com este método, a partição dos dados de entrada é baseada na suposição de que o *software* a ser testado se comporta da mesma forma para um grupo de dados que formam uma classe de partição. A idéia é dividir um domínio de entrada para um número finito de subdomínios e escolher um dado modelo de teste a partir de cada um desses subdomínios. Assim, os dados são particionados de acordo com alguns critérios que incidirão sobre as características das entradas de dados que possam ter um efeito sobre o comportamento do *software* que está sendo testado. O método de “particionamento por categoria” é adotado em diversas pesquisas que realizam testes de transformações de modelo, no âmbito de MDA (FLEUREY *et al*., 2009; PONS e GARCIA, 2008; LAMARI, 2007).

A abordagem para a geração de modelos significativos (FLEUREY *et al*., 2009), baseada na topologia do domínio a ser particionado, também foi utilizada nesta tese em complementação ao método de "particionamento por categoria". Em consequência, o particionamento foi realizado com base na topologia definida nos perfis de ligação, utilizados para marcar o modelo PIM.

Com base nas abordagens do "particionamento por categoria" e da "geração de modelos significativos", os critérios de cobertura foram definidos e utilizados para selecionar os casos de teste. De um modo geral, estes critérios se baseiam em elementos do programa a serem exercitados. A cobertura mede o grau em que os elementos de uma implementação são executados durante os testes. No contexto desta pesquisa, que utiliza a linguagem de transformação de modelos ATL (baseada em regras) e que é baseada em um perfil de plataforma que contém diversos estereótipos (organizados como um modelo de domínio), um conjunto de modelos de teste adequado para as transformações desenvolvidas deve ao menos cumprir os seguintes requisitos de cobertura:

- *Cobertura por regras*: cada regra de transformação definida deve ser exercitada por pelo menos um modelo de teste. Dado que a ATL é uma linguagem de transformação baseada em regras, a cobertura por regras é adequada neste contexto.
- *Cobertura por elemento de ligação*: cada elemento de ligação definido por meio de um estereótipo deve ser instanciado em pelo menos um modelo de teste. Desse modo, a idéia é isolar os elementos definidos nesses perfis em faixas específicas, a fim de assegurar que esses valores especiais serão usados para o teste.

Visto que, no contexto desta pesquisa, perfis (metamodelos) de plataforma e de ligação definem o espaço dos modelos de entrada das transformações, definir critérios baseados nesses metamodelos e utilizá-los para a partição do espaço da entrada, com o objetivo de selecionar um conjunto de dados de teste relevantes, é adequado para a geração de casos de teste (WANG *et al*., 2008). Desse modo, a seleção dos casos de testes envolveu a seleção de modelos de entrada (PIMs) válidos a partir de um conjunto de modelos de entrada. Estes modelos de teste são válidos no sentido de que eles pertencem ao domínio de entrada da transformação (em conformidade com metamodelo de entrada) (SEN *et al*., 2008).

As próximas subseções descrevem a criação do conjunto de casos de teste para a transformação MT-AMP e MT-PROAPES, implementadas durante a realização desta pesquisa.

4.10.2 Criação do conjunto de casos de teste para a transformação MT-AMP

Considerando a transformação de modelos MT-AMP desenvolvida nesta pesquisa, a criação do conjunto de casos de teste envolveu a seleção de modelos que atendam aos seguintes critérios de cobertura:

- *Cobertura por regras*: foram consideradas todas as regras que compõe o modulo MT-AMP, sendo elas: *Operation*, *OperationStereotype*, *OperationRtSw*, *OperationRtDD* e *Model*.
- *Cobertura por elementos de ligação*: o perfil de ligação AMP define dois estereótipos utilizados para a marcação dos modelos PIMs e que devem ser considerados para a criação dos modelos de teste: *rtSwOperation* e *rtDDOperation*.

4.10.3 Criação do conjunto de casos de teste para a transformação MT-PROAPES

A seleção do conjunto de casos de teste para a transformação de modelos MT-PROAPES envolveu a seleção de modelos conforme os seguintes critérios de cobertura: cobertura por regras e cobertura por elementos de ligação.

- *Cobertura por regras*: foram consideradas todas as regras que compõem o módulo MT-PROAPES, listadas na Tabela 9.
- *Cobertura por elemento de ligação*: o perfil de ligação *dynRTOS* é composto pelos seguintes subperfis *dynSwRTOS* e *dynDDRTOS*. O subperfil *dynSwRTOS* define os seguintes estereótipos utilizados para a marcação de mensagens em diagramas de sequência e que devem ser considerados para a criação dos modelos de teste: *dynActionSwRTOS*, *dynCheckMessage*, *dynInitSched*, *dynSendMsg*, *dynReceiveMsg*, *dynActivateThread*, *dynInitRTOS*, *dynSleepThreadTime* e, *dynGetIdThread*. Por sua vez, subperfil *dynDDRTOS* define os seguintes estereótipos utilizados para a marcação de modelos PIMs e que devem ser considerados para a seleção dos modelos de teste: *dynStartDD*, *dynRegisterReceiverDD*, *dynSendCommandDD*, *dynGoToLineColDD*, *dyxPosicDD* e *dynWriteStringDD*.

4.10.4 Execução dos testes

Os testes das transformações de modelos baseadas em ATL foram executados considerando-se os conjuntos de casos de teste selecionados. Para tal, é interessante detalhar a execução de uma transformação ATL, que envolve os seguintes passos: i) verificação de erros semânticos com relação ao metamodelo ATL e aos metamodelos de origem e de destino; ii) compilação; e iii) execução da transformação (JOUAULT e KURTEV, 2006). Desta forma, nos dois primeiros passos a verificação de certos tipos de erros já é realizada e a execução é interrompida, quando necessário. A corretude sintática da transformação de modelos é garantida pelo processo de compilação da linguagem ATL. Desse modo, o compilador ATL verifica se o programa/módulo desenvolvido está sintaticamente correto com relação a essa linguagem. Durante a execução da transformação, as variáveis são inicializadas e, novamente neste momento, quando erros são encontrados a transformação é interrompida.

Erros de sintaxe encontrados nesta fase foram resolvidos durante a realização dos referidos testes. Outros casos que apresentaram erro foram aqueles em que as propriedades (*tagged-values*) relacionadas aos estereótipos não haviam sido corretamente preenchidas pelo usuário. Ainda, pertinente ressaltar que este erro não se reflete no modelo alvo gerado pela transformação, pois o processo de execução da transformação é interrompido pela ATL no momento em que o erro é detectado. Para tentar minimizar este problema, restrições OCL podem ser definidas, por meio de regras OCL, tornando possível validar o modelo PIM com relação às restrições OCL definidas. Desse modo, é possível se antecipar ao problema, garantindo que o PIM esteja definido em conformidade com os requisitos exigidos pela transformação. A criação de tais restrições OCL é proposta como sugestão de trabalho futuro relacionado com esta pesquisa.

Outros erros encontrados e solucionados estavam relacionados com a definição das regras de transformação. Dado que as regras de transformação, em ambas as transformações (MT-AMP e MT-PROAPES), são fortemente relacionadas com os estereótipos definidos nos perfis de ligação (AMP e *dynRTOS*), os comandos definidos devem recuperar os valores das propriedades (*tagged-values*) relacionadas a esses estereótipos. Nesse momento, em algumas regras o nome das propriedades não estava em conformidade com o estereótipo definido, ou seja, em alguns casos a *tagged-value* definida em uma regra que acessa elementos no contexto de um dado estereótipo estavam acessando propriedades relacionadas com outro estereótipo do mesmo perfil.

4.11 CONCLUSÕES DO CAPÍTULO

Este capítulo apresentou o método PI-MT para criação de transformações de modelo, fazendo uso de modelos de plataforma explicitamente definidos. O método proposto visa a sistematizar o processo de criação de transformações de modelos reutilizáveis e adaptáveis a diferentes plataformas baseadas em RTOS. O PI-MT apoia a construção e evolução de tecnologias de *software* embarcado, particularmente no que diz respeito ao uso de sistemas operacionais em tempo real.

O método PI-MT incorpora as transformações MT-AMP e MT-PROAPES, baseadas no RTOS X *Real-Time Kernel*. A transformação de modelos MT-AMP visa ao mapeamento de conceitos genéricos de RTOS em uma plataforma específica que consiste de um RTOS relacionado com uma plataforma de *hardware* baseada em um processador específico. A MT-AMP tem como foco diagramas de classes e permite o reuso das mesmas regras de transformação para diferentes plataformas por meio de um Modelo de Plataforma explicitamente definido. O uso de PMs explicitamente definidos traz relevantes contribuições para o desenvolvimento de *software* embarcado em termos da independência entre modelos e plataformas.

A transformação proposta neste capítulo foi implementada utilizando-se a linguagem *Atlas Transformation Language* (ATL). A linguagem ATL é amplamente reconhecida como uma solução para transformações de modelos no contexto da MDA. Também, a implementação realizada se beneficia das principais vantagens da técnica de sobreposição, sendo elas: a melhoria da manutebilidade e o aumento da reusabilidade.

Este capítulo também apresentou a transformação de modelos PIM-comportamental em PSM comportamental, denominada MT-PROAPES, que utiliza o perfil PROAPES e que tem como foco diagramas de sequência. A principal contribuição da transformação proposta é permitir o reuso das regras de transformação para diferentes versões de uma determinada plataforma de RTOS, tendo como base diagramas comportamentais dos sistemas de *software* embarcado.

As transformações MT-AMP e MT-PROAPES definem módulos de refinamento (PIM2PSM.atl e MT-PROAPES.atl, respectivamente) que sobrepõe o módulo UML2Copy.atl. O módulo UML2Copy.atl realiza a cópia de um modelo UML e é amplamente utilizado em pesquisas de transformação de modelos (GRONMO *et al*., 2009, VAN AMSTEL e VAN DEN BRAND, 2011).

Para o futuro, é possível ampliar o método PI-MT por meio do desenvolvimento de uma transformação de modelos que suporte modelos PIM representados na forma de diagramas de estado. O escopo da tese foi limitado, atendendo a diagramas de classe e de sequência, visando comprovar a teoria. Entretanto, o PI-MT pode ser estendido para operar em outros diagramas UML, como o diagrama de estados.

A técnica de teste de *software* denominada "*caixa preta*", utilizada para testar as transformações MT-AMP e MT-PROAPES, também foi apresentada neste capítulo. Esta técnica possibilitou testar o funcionamento das transformações implementadas, permitindo avaliar se o resultado produzido era o resultado esperado. No futuro, técnicas mais avançadas podem ser estudadas e selecionadas para realizar a verificação das transformações implementadas por meio do método proposto.

5 CASO DE ESTUDO

Este capítulo apresenta um caso de estudo realizado durante o desenvolvimento desta tese no qual um sistema embarcado foi implementado. O sistema embarcado desenvolvido foi um Simulador de Alarme (SA), projetado para duas plataformas de *hardware* diferentes: uma placa de avaliação eSysTech eAt55 e uma placa de avaliação eSysTech eLPC48. Ainda, por meio desse caso de estudo foi possível exemplificar a integração do método *Platform Independent - Model Transformations* (PI-MT) à abordagem MDA.

5.1 CASO DE ESTUDO – SISTEMA SIMULADOR DE ALARME

Nesta seção, são apresentados os objetivos do caso de estudo, a descrição das características do sistema projetado, a organização do *software*, a descrição da plataforma de *hardware* e *software* na qual o sistema está inserido, o ambiente de desenvolvimento utilizado e a aplicação do método PI-MT ao caso de estudo. Esse caso de estudo foi desenvolvido juntamente com a pesquisadora Inali Wisniewski Soares, integrante do mesmo grupo de pesquisa da UTFPR.

5.1.1 Objetivos do Caso de Estudo

O sistema desenvolvido no caso de estudo é um Simulador de Alarme (SA). Os principais objetivos desse caso de estudo foram verificar o método PI-MT e permitir a realização de testes nas transformações de modelos implementadadas com base nos modelos de *software* criados. Como objetivos específicos deste caso de estudo têm-se:

1. Desenvolver um sistema embarcado utilizando a abordagem tradicional de desenvolvimento de *software*;
2. Desenvolver um sistema embarcado utilizando a abordagem de desenvolvimento de *software* dirigida a modelos, com base no método PI-MT;
3. Desenvolver habilidades no desenvolvimento de sistemas embarcados utilizando a linguagem C++;

4. Desenvolver habilidades no desenvolvimento de sistemas embarcados utilizando o núcleo de tempo real, denominado RTOS *X Real-Time Kernel*;
5. Desenvolver o SA baseado no RTOS *X-Real Time Kernel* para duas plataformas de *hardware* distintas*:* placa eSysTech eAt55 e placa eSysTech eLPC48.

5.1.2 Descrição do Sistema Simulador de Alarme

A aplicação desenvolvida para o caso de estudo consistiu em um *software* de sistema embarcado para o sistema Simulador de Alarme (SA). O SA simula o funcionamento de um alarme e tem como função principal realizar o monitoramento de um ambiente com o objetivo de detectar a presença de intrusos. O menu do sistema possui as seguintes opções: 1) Ativar alarme; 2) Desativar alarme; 3) Alterar a senha e 4) Sair.

O alarme pode ser ativado ou desativado por meio de um teclado com cinco teclas. Ao usar o teclado para a ativação ou a desativação do alarme, uma senha com cinco dígitos deve ser inserida. O sistema permite a alteração deste código de acesso. Um tempo de espera foi programado para permitir ao proprietário do imóvel a ativação ou a desativação do alarme, sem que o alarme dispare.

Quando o alarme está ativado e a presença de um intruso é detectada, o alarme deve emitir um *beep* sonoro, que atualmente está sendo simulado por meio dos LEDs do alarme.

Uma rotina simula uma invasão ao sistema e envia aleatoriamente sinais de alerta de intruso. Quando o alarme é ativado, existe um tempo em que o alarme não detecta a presença de intrusos, para permitir a saída do proprietário.

O alarme possui oito LEDs vermelhos utilizados para representar o estado do alarme. O alarme está sendo ativado quando o primeiro e o último LED piscam simultaneamente. O alarme está ativo quando o primeiro LED está piscando. O alarme está reagindo a um movimento detectado pelo sensor quando os LEDs 0, 2, 4 e 6 piscam simultaneamente. Quando todos os LEDs estão piscando em sequência indicam que a sirene está soando, i.e., o alarme está reagindo à detecção de um movimento.

5.1.3 Organização do *Software*

O sistema operacional tem como base o uso de *threads* para executar a aplicação e, para tanto, utiliza um algoritmo de escalonamento tipo *Round-Robin* que organiza a execução das *threads* de forma sequencial a partir de sua ordem de criação, considerando seu nível de prioridade.

Para atender às necessidades do escopo do sistema embarcado proposto, são necessárias diversas rotinas: gerenciar senha, gerenciar menu, gerenciar alarme, gerenciar sirene, gerenciar *display*, gerenciar LEDs, gerenciar receptor de interação e gerenciar sensor. Para serem executadas, essas rotinas dependem da troca de mensagens entre os objetos de controle do sistema. Esses objetos são instanciados pelas funções de inicialização das *threads* e possuem execução infinita, mantendo suas *threads* ativas e prontas para reagir às mensagens recebidas.

A Tabela 10 apresenta as *threads* definidas para o SA. Essas *threads* são reativas, executando rotinas em resposta às mensagens recebidas. Uma exceção é a *thread* sensor, que emula um sensor de movimento, disparando sinais periódicos para o alarme.

5.1.4 Arquitetura de *Hardware* e Processadores para o SA

O SA foi projetado para duas plataformas de *hardware* distintas: placa eSysTech eAt55 *e* placa eSysTech eLPC48. A eAT55 é uma placa de avaliação produzida pela eSystech (ESYSTECH, 2012) para a arquitetura *Advanced RISC Machines* (ARM) (ARM, 2010). Na Figura 50, a placa eAT55 é ilustrada juntamente com um visor de cristal líquido (*Liquid Crystal Display* – LCD) de 4 linhas e 20 colunas e um teclado simples com 5 teclas.

Placas de avaliação, como a placa eAT55, são usadas como plataforma para experimentação, aprendizado, avaliação e treinamento no desenvolvimento de sistemas embarcados em várias organizações, tais como: departamentos de engenharia, laboratórios de pesquisa, universidades, escolas e centros de treinamento (ESYSTECH, 2012). A placa eAT55 usa um processador AT91M5580 da Atmel (ATMEL, 2010) com núcleo ARM7TDMI de 32 bits, baseado no modelo de arquitetura *Reduced Instruction Set Computer* (RISC) (ESYSTECH, 2012).

Tabela 10 – *Threads* do SA

***Thread* Alarme**	
Definição	Controlar o estado do alarme do sistema. Recebe mensagens das *threads* sensor e menu. Envia mensagens para as *threads* LEDS, *display* e sirene.
Funções	a) ativar o alarme; b) desativar o alarme; c) reagir a um intruso detectado.
***Thread* Senha**	
Definição	Controlar o estado do gerenciador de senhas do sistema. Recebe mensagens da *thread* menu. Envia mensagens para a *thread* menu.
Funções	a) alterar senha; b) verificar senha.
***Thread* Display**	
Definição	Criar um objeto do tipo *display* para controlar o estado do *display* do sistema de alarme. Recebe mensagens das *threads* alarme, sirene e menu.
Funções	a) exibir o estado do alarme no *display* (“ativando”, “ativo, “inativo” e “reagindo”); b) exibir o estado da sirene no *display* (“ativa” e “inativa”); c) exibir o menu específico do alarme no display (“1. Ativar Alarme”, “2. Desativar Alarme”, “3. Alterar a Senha”, “4. Sair”).
***Thread* Sirene**	
Definição	Criar um objeto do tipo sirene para controlar o estado da sirene do sistema. Recebe mensagens da *thread* alarme. Envia mensagens para as *threads* LEDs e *display*.
Funções	a) ativar sirene; b) Desativar sirene.
***Thread* LEDs**	
Definição	Controlar o estado dos LEDs do sistema. Recebe mensagens das *threads* alarme e sirene.
Funções	a) emular alarme ativando; b) emular alarme ativo; c) emular alarme reagindo; d) emular alarme inativo; e) emular sirene.
***Thread* Menu**	
Definição	Controlar o estado do menu do sistema. Recebe mensagens das *threads* receptor de interação e Senha. Envia mensagens para as *threads* Display, Senha e Alarme.
Funções	a) selecionar opção do menu; b) bloquear menu; c) desbloquear menu.
***Thread* ReceptorInteração**	
Definição	Controlar os sinais recebidos do teclado.
Funções	a) enviar mensagens para a *thread* Menu.
***Thread* Sensor**	
Definição	Controlar o estado do sensor do sistema.
Funções	a) Envia mensagens para a *thread* alarme.

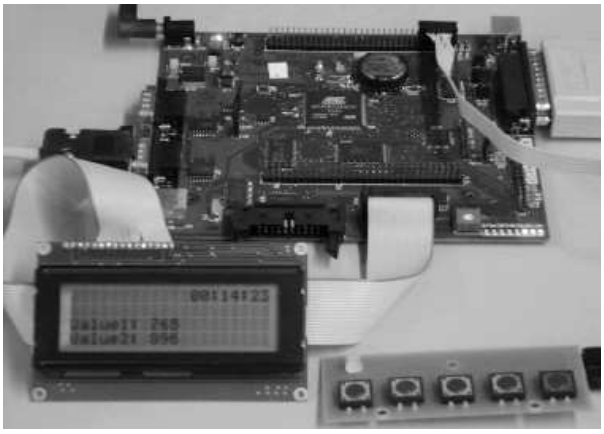

Figura 50 – Placa eAT55 - eSysTech

Por sua vez, o módulo eLPC48 é um *System On Module* (SOM) baseado em processadores ARM da série LPC21xx da NXP. Na Figura 51, o módulo eLPC48 é ilustrado juntamente com um visor de cristal líquido (*Liquid Crystal Display* – LCD) de 4 linhas e 20 colunas e um teclado simples com 5 teclas

Esse módulo foi projetado e é fabricado pela eSysTech como uma das alternativas de sua família de placas SOM. Esse módulo consiste de uma placa de circuito impresso (*Printed Circuit Board* - PCB) da dimensão de meio cartão de crédito (*half credit card size*) que implementa o *core* de um sistema microprocessado baseado na arquitetura ARM7TDMI-S. O módulo eLPC48 implementa a funcionalidade essencial (processador, alimentação e *clock*) de um sistema embarcado e disponibiliza uma interface padrão para a implementação de funcionalidades específicas a cada produto em uma placa base. Esse módulo pode ser empregado para fins de engenharia ou para integração a um produto final (ESYSTECH, 2012).

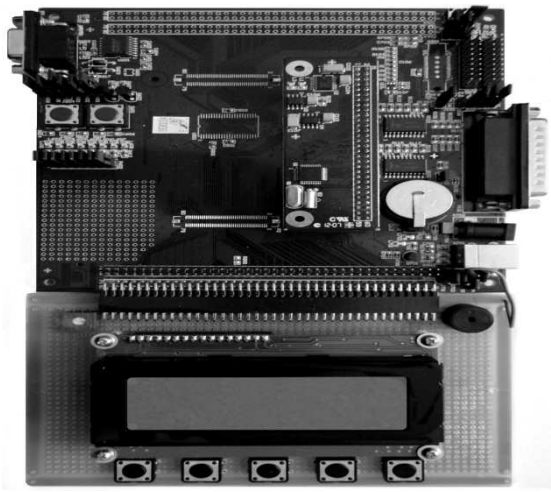

Figura 51 – Placa eLPC48 – eSysTech

5.1.5 Projeto do SA - Abordagem Tradicional de Desenvolvimento de *Software*

O Sistema SA foi, primeiramente, desenvolvido com base na abordagem tradicional de desenvolvimento de *software*. Desse modo, os modelos do sistema foram criados em UML e o código-fonte descrito na linguagem C++. O mesmo sistema foi projetado e implementado

para duas plataformas distintas (X - eAt55 e X - eLPC48), citadas na subseção anterior (5.1.4).

Visto que, na abordagem tradicional de desenvolvimento de *software* não existe uma separação entre as funcionalidades do *software* e as características de plataforma, neste projeto os elementos relacionados com as plataformas de implementação foram definidos diretamente no PIM. Em adição, o código do sistema foi definido em conformidade com os modelos que integravam o PIM, mas em etapas distintas de desenvolvimento. O detalhamento do projeto do SA, utilizando a abordagem tradicional de desenvolvimento de *software*, pode ser encontrado em (AGNER, 2010).

5.1.6 Projeto do SA - Abordagem Dirigida a Modelos

O sistema SA foi implementado com base na abordagem de desenvolvimento de *software* dirigida a modelos, por meio da aplicação do método PI-MT, proposto nesta tese. A etapa inicial do método PI-MT é a definição dos módulos de transformação de modelos a serem utilizados.

Esse caso de estudo realizou a modelagem estrutural e comportamental do SA, por meio de diagramas de classe e de diagramas de sequência, respectivamente. Desse modo, foram definidos e selecionados os módulos MT-AMP e MT-PROAPES (ambos definidos no Capítulo 4). O primeiro suporta a transformação de diagramas de classes e o último trata de transformações baseadas em diagramas de sequência.

Os PMs utilizados nesse caso de estudo foram construídos por Soares (2012) com base no perfil PROAPES (definido no Capítulo 4). Estes PMs representam a plataforma formada pelo núcleo de tempo real desenvolvido pela eSysTech denominado X *Real-Time Kernel* e empregado placas eAT55 e eLPC48. Esse RTOS é uma camada de *software* que oferece serviços compartilhados de gerenciamento de tarefas, de sincronização, de temporização e de troca de mensagens (GOES; RENAUX, 2006). Além disso, esse RTOS disponibiliza formas de interagir com periféricos e/ou recursos de *hardware* por meio de drivers de dispositivos. Os PMs são construídos por meio de diagramas de classes *UML* e as classes são agrupadas de acordo com as características de *software* e de *hardware* em comum.

O sistema SA possui vários modelos de casos de uso, entretanto, a fim de reduzir a exemplificação do sistema, optou-se por apresentar apenas os modelos de casos de uso que apresentam serviços e atributos do RTOS. Desse modo, é possível visualizar o uso da

integração do PI-MT à abordagem MDA que é um dos principais objetivos desse caso de estudo. Além disso, os modelos UML que serão ilustrados para o SA são: os modelos de casos de uso, os modelos de sequência e o modelo de classes. Assim, foi possível reduzir o tamanho do caso de estudo e ilustrar a definição e a transformação de modelos estáticos e dinâmicos, no contexto da MDA.

Inicialmente são apresentados os requisitos identificados para o SA e que são necessários para o desenvolvimento do PIM desse sistema:

- O sistema deverá controlar a ativação e a desativação de um alarme. A ativação e a desativação são acionadas por um usuário por meio de um teclado.
- O sistema deverá permitir a inserção e alteração de um código de cinco dígitos para ativar e desativar o alarme.
- A senha deve possuir cinco dígitos, e cada dígito deve estar entre os valores {0,1,2,3,4}.
- Estando o alarme ativado e a presença de um intruso detectada, o alarme deve emitir um *beep* sonoro, que atualmente está sendo simulado por meio dos LEDs do alarme.
- Uma vez ativado o alarme, o sistema determina um tempo em que o alarme não detecta intruso, para permitir a saída do proprietário.
- O alarme tem oito LED´s vermelhos. O primeiro e o último LED piscando simultaneamente indicam que o alarme está sendo ativado. O primeiro LED piscando indica que o alarme está ativo. Os LED’s 0, 2, 4 e 6 piscando, simultaneamente, indicam que o alarme está reagindo a um movimento detectado pelo sensor do alarme. Quando todos os LEDs estão piscando em sequência, indicam que a sirene está soando, pois o alarme está reagindo a detecção de um movimento.
- O sistema possui um menu que realiza a interface com o proprietário e apresenta as seguintes opções: 1. Ativar alarme, 2. Desativar alarme, 3. Alterar a senha e 4. Sair.

Com base nos requisitos definidos para o sistema SA os modelos de casos de uso foram especificados. A Tabela 11 apresenta os atores identificados para o sistema SA.

Os pacotes de modelos de casos de uso definidos para o SA foram os seguintes: i) Comum; ii) Inicializar Sistema de Alarme; iii) Gerenciar LEDs; iv) Gerenciar Alarme; v)

Gerenciar Display; vi) Gerenciar Menu; vii) Gerenciar Sensor; viii) Gerenciar Sirene; e ix) Gerenciar Senha. Esses pacotes são detalhados a seguir.

Tabela 11 – Atores do SA

Ator	**Papel**
Mensageiro	envia, recebe e checa mensagens enviadas pelas *threads* do sistema
OS	gerencia as *threads* do sistema
Alarme	gerencia o alarme
Menu	gerencia o menu
Controlador	inicializa o *RTOS*, o escalonador das *threads* e as *threads* do sistema
Keyboard	gerencia o teclado
LCD	gerencia o *display*
LED	gerencia os LEDs
Sirene	gerencia a sirene
Sensor	gerencia o sensor

O pacote Comum refere-se às funcionalidades comuns do sistema e é usado pelos demais modelos de casos de uso do sistema. Esse pacote, ilustrado na Figura 52, é composto pelos seguintes modelos de casos de uso:

- Enviar mensagens entre as *threads*: efetua o envio de mensagens entre as *threads* do SA.
- Receber mensagens entre as *threads*: realiza o recebimento das mensagens enviadas pelas *threads* do SA.
- Checar mensagens: efetua a checagem de mensagens entre as *threads* do SA.

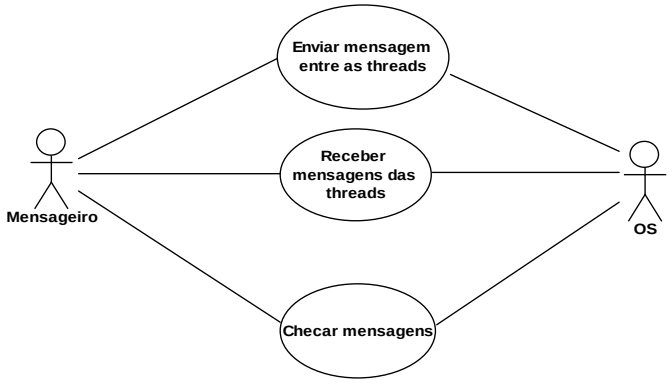

Figura 52 – Modelo de Casos de Uso "Comum"

O pacote "Inicializar Sistema de Alarme" é ilustrado na Figura 53 e refere-se às funcionalidades de inicialização do sistema Simulador de Alarme.

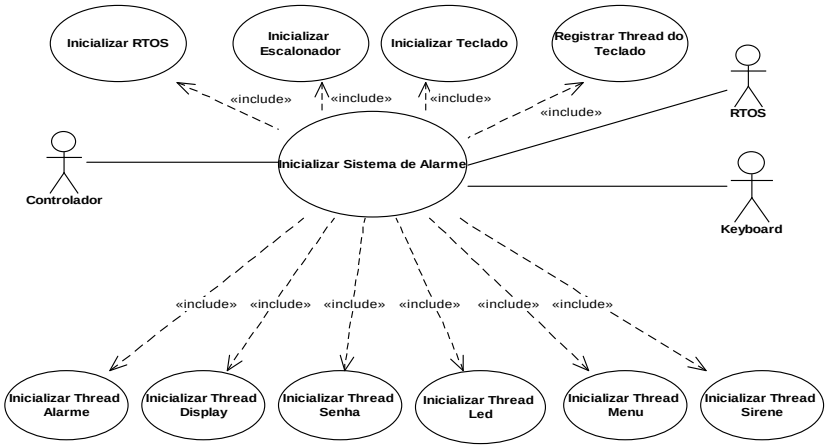

Figura 53 – Modelo de Casos de Uso "Inicializar Sistema de Alarme"

O pacote "Gerenciar LEDs" refere-se às funcionalidades de gerenciamento de LEDs do sistema SA e é ilustrado na Figura 54.

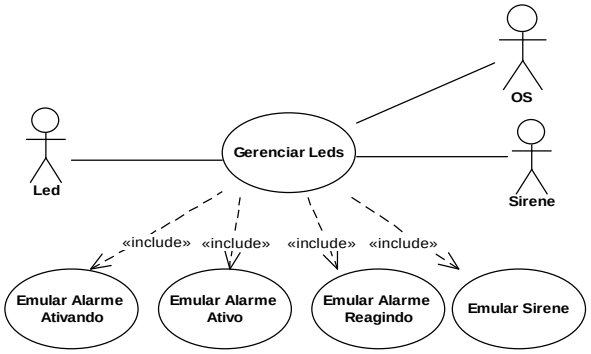

Figura 54 – Modelo de Casos de Uso "Gerenciar LEDs"

O pacote "Gerenciar Alarme" refere-se às funcionalidades de gerenciamento do alarme e é ilustrado na Figura 55.

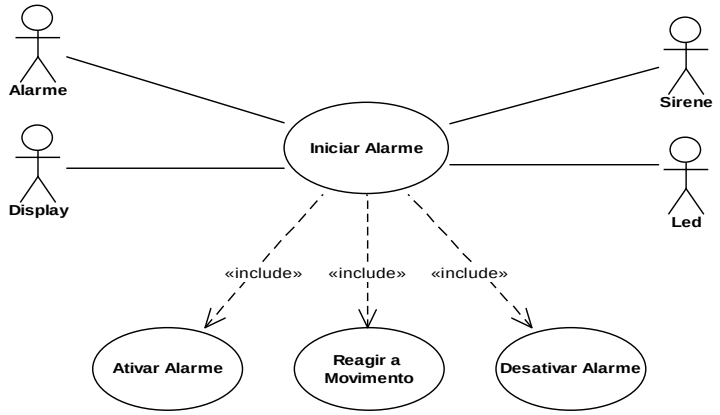

Figura 55 – Modelo de Casos de Uso "Gerenciar Alarme"

O pacote "Gerenciar *Display*", ilustrado na Figura 56, refere-se às funcionalidades de gerenciamento do *display*.

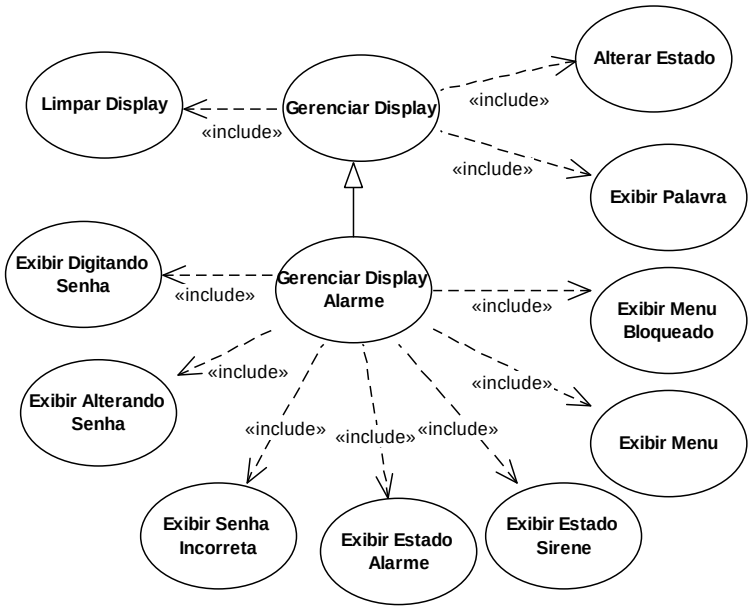

Figura 56 – Modelo de Casos de Uso "Gerenciar *Display*"

O pacote "Gerenciar Menu" é ilustrado na Figura 57 e refere-se às funcionalidades de gerenciamento do menu.

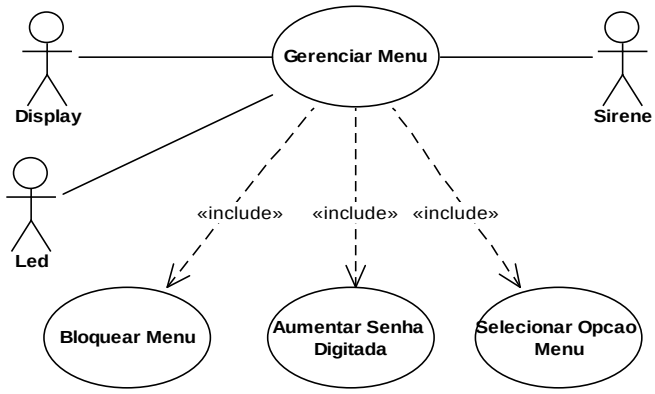

Figura 57 – Modelo de Casos de Uso “Gerenciar Menu”

O pacote “Gerenciar Sensor” é ilustrado na Figura 58 e refere-se às funcionalidades de gerenciamento do sensor.

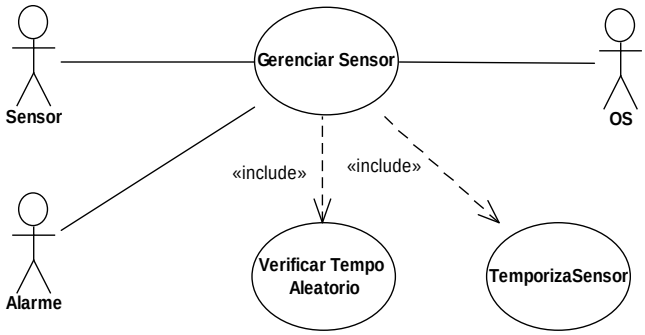

Figura 58 – Modelo de Casos de Uso “Gerenciar Sensor”

O pacote “Gerenciar Sirene” refere-se às funcionalidades de gerenciamento da sirene e é ilustrado na Figura 59.

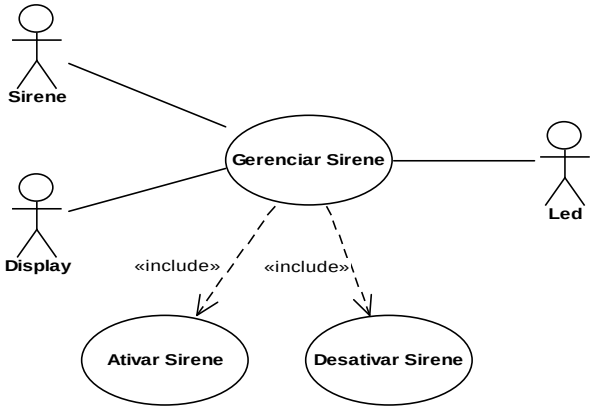

Figura 59 – Modelo de Casos de Uso “Gerenciar Sirene”

Por fim, o pacote “Gerenciar Senha” refere-se às funcionalidades de gerenciamento de senha do SA e é ilustrado na Figura 60.

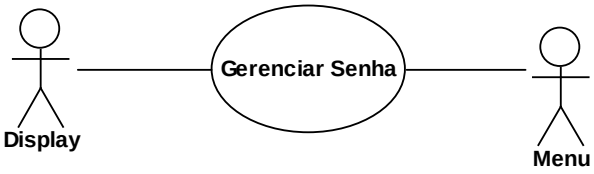

Figura 60 – Modelo de Casos de Uso “Gerenciar Senha”

As próximas subseções ilustram e descrevem os modelos de sequência definidos para os modelos de casos de uso do SA. Além disso, a última subseção ilustra e descreve o modelo de classes para o sistema SA. Por meio desses modelos é possível exemplificar a aplicação do método PI-MT.

5.1.6.1 Modelos de Sequência para o Pacote “Comum”

A Figura 61 exemplifica uma transformação de modelos baseada no módulo MT-PROAPES e refere-se ao modelo de sequência (PIM) definido para o modelo de caso de uso “Comum”: “Enviar mensagens entre as *threads*”. O pacote PM - RTOS X - eAt55, ilustrado nessa figura, apresenta a classe “X” cujo estereótipo <<swxCore>> definido no perfil

PROAPES indica que essa classe representa os conceitos básicos de descrição de contextos de *software* de execução concorrente.

O estereótipo <<swxCore>> está anotado na classe “X” e é suficiente para atender todos os requisitos dessa classe, ou seja, os serviços definidos nessa classe possuem valores rotulados correspondentes definidos no estereótipo <<swxCore>>, possibilitando a definição de semântica para os elementos de modelos da classe “X”.

A nota ligada à classe “X” mostra a atribuição de valores rotulados, definidos no estereótipo <<swxCore>>, para os serviços da classe “X”, por exemplo, *sendPutMsg* revela que *Put* representa o envio assíncrono de uma mensagem no formato *PutMsg* (estrutura de dados definida no perfil PROAPES) e *getTIdThread* revela que o serviço *GetTid* informa o identificador de uma tarefa.

O PIM, ilustrado na Figura 61, mostra o uso do perfil dinâmico *dynSwRTOS* nas seguintes mensagens:

- *ObterIdThread* - anotada pelo estereótipo <<dynGetIdThread>> (revela que essa mensagem representa a ação de identificação de uma tarefa) e que possui os parâmetros: *nomeThread* que indica o nome da *thread* e *idThread* que indica o identificador da *thread*;
- *EnviarMensagem* - anotada pelo estereótipo <<dynSendMsg>> (revela que essa mensagem representa o envio de uma mensagem assíncrona) e que possui os parâmetros: *idThread* e *mensagem* que representam o identificador da *thread* e a mensagem enviada, respectivamente. As notas são anexadas ao diagrama de sequência para ilustrar o uso dos valores rotulados associados às mensagens do PIM.

A vinculação entre PIM e PM é realizada por meio dos valores rotulados definidos no estereótipo «*dynActionSwRTOS*». Esse estereótipo representa as ações gerais de *software* em projetos baseados em RTOS e estende a metaclasse *Message* que descreve as mensagens trocadas entre os objetos e entidades envolvidas em um sistema (usada em diagramas de sequência).

Os estereótipos «*dynGetTIdThread*» e «*dynSendMsg*» especializam o estereótipo «*dynActionSwRTOS*». Esse estereótipo define valores rotulados usados para ligar o PM (anotado com os estereótipos do perfil *swxRTOS*) ao PIM (anotado com os estereótipos do perfil *dynRTOS*).

Os seguintes valores rotulados são definidos pelo estereótipo «*dynActionSwRTOS*»:

- *opSwTarget*: indica a propriedade definida no estereótipo do perfil *swxRTOS* que corresponde a uma ação definida no estereótipo «*dynActionSwRTOS*».
- *paSwTarget*: indica as propriedades definidas no estereótipo do perfil *swxRTOS* que correspondem aos parâmetros definidos no estereótipo «*dynActionSwRTOS*».
- *rtSwService*: indica o nome do perfil responsável por definir o PM abstrato que será associado ao PIM definido no perfil *dynSwRTOS.*

Os modelos PM e PIM são inseridos como atributos de entrada no módulo MT-PROAPES responsável por realizar a transformação do PIM em um PSM X - eAt55. A transformação MT-PROAPES, resumidamente ilustrada na Figura 61, usa um PIM marcado com os estereótipos do perfil *dynRTOS* e o PM marcado com os estereótipo do perfil *swxRTOS*.

Como resultado, a ligação entre os elementos marcados do PIM e os respectivos elementos do PM é realizada pela transformação MT-PROAPES com base nessas propriedades. Desse modo, a transformação busca por cada elemento marcado do PIM, encontra os respectivos serviços de RTOS no PM com base nos valores das propriedades e, então, gera o modelo PSM contendo os detalhes específicos da plataforma de execução considerada: RTOS *X Real-Time Kernel* e placa eAt55.

Neste exemplo, as principais diferenças identificadas entre o PIM e o PSM (gerado por meio módulo MT-PROAPES) são as seguintes: i) no PSM gerado os estereótipos (do perfil de ligação) não são aplicados nos elementos de modelos; e ii) as mensagens e respectivos parâmetros do PIM foram transformadas para as mensagens e respectivos parâmetros usadas de acordo com o modelo de plataforma selecionado, por exemplo, a mensagem *ObterIDThread* foi transformada para *GetTId*.

As Figuras 62 e 63 ilustram a transformação de modelos com base no módulo MT-PROAPES para os seguintes modelos de sequência definidos para os modelos de casos de uso comum: “Receber mensagens entre as *threads*” e “Checar mensagens”, respectivamente. Esses modelos de sequência ilustram o uso de serviço RTOS, porém não são descritos em detalhes, por apresentarem características semelhantes ao exemplo ilustrado na Figura 61.

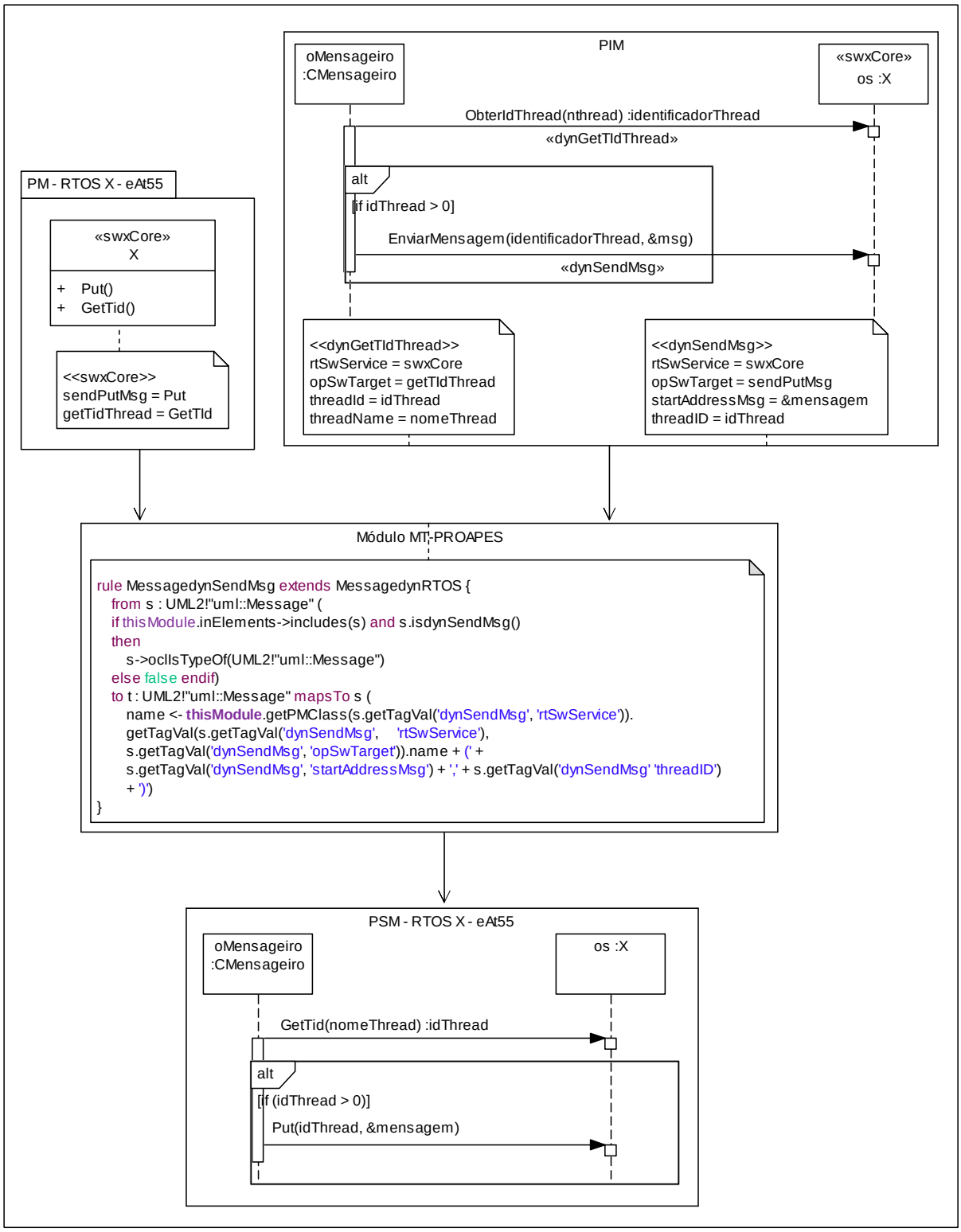

Figura 61 – Modelo de Sequência "Enviar mensagem entre as *threads*"

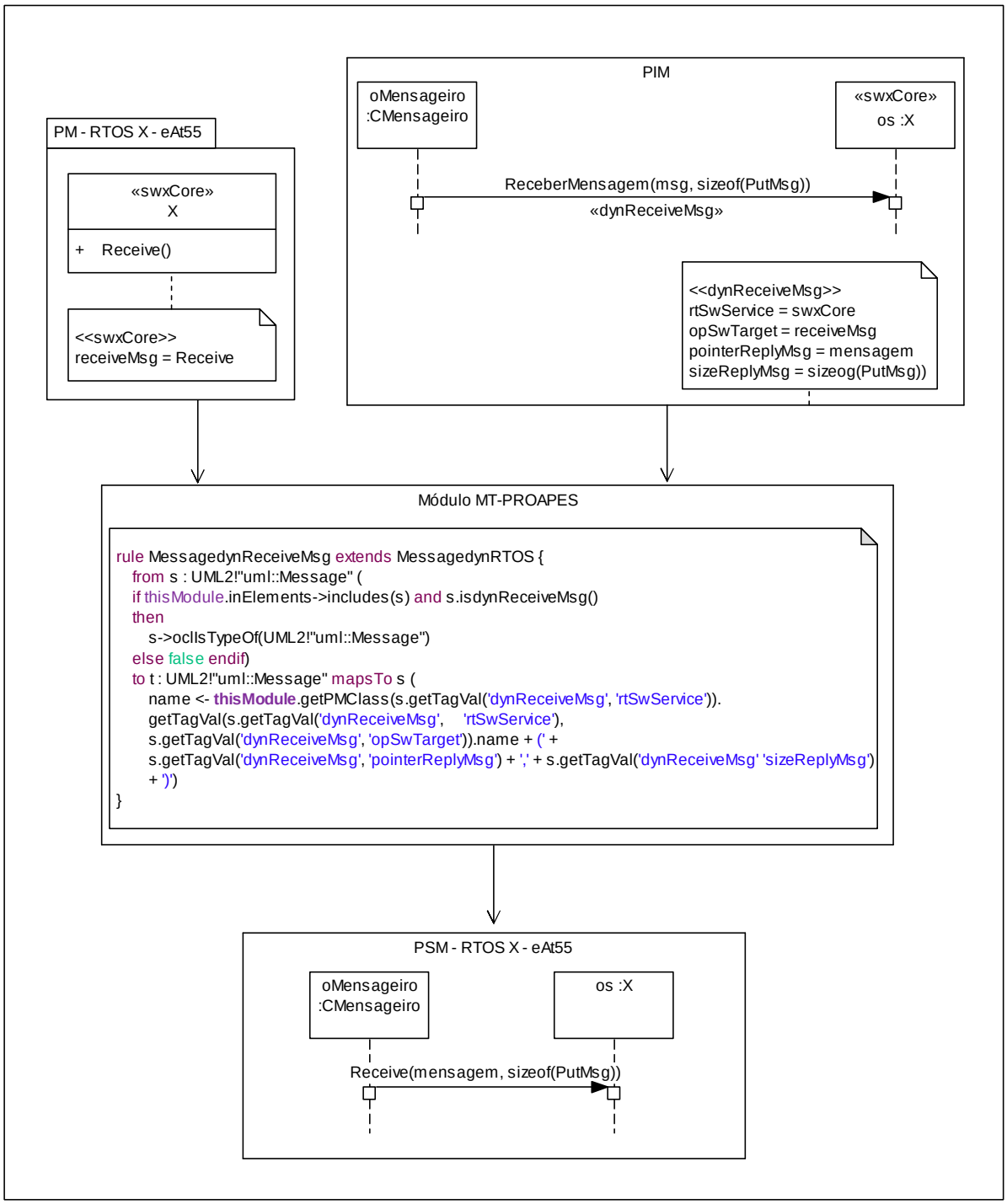

Figura 62 – Modelo de Sequência "Receber mensagem entre as *threads*"

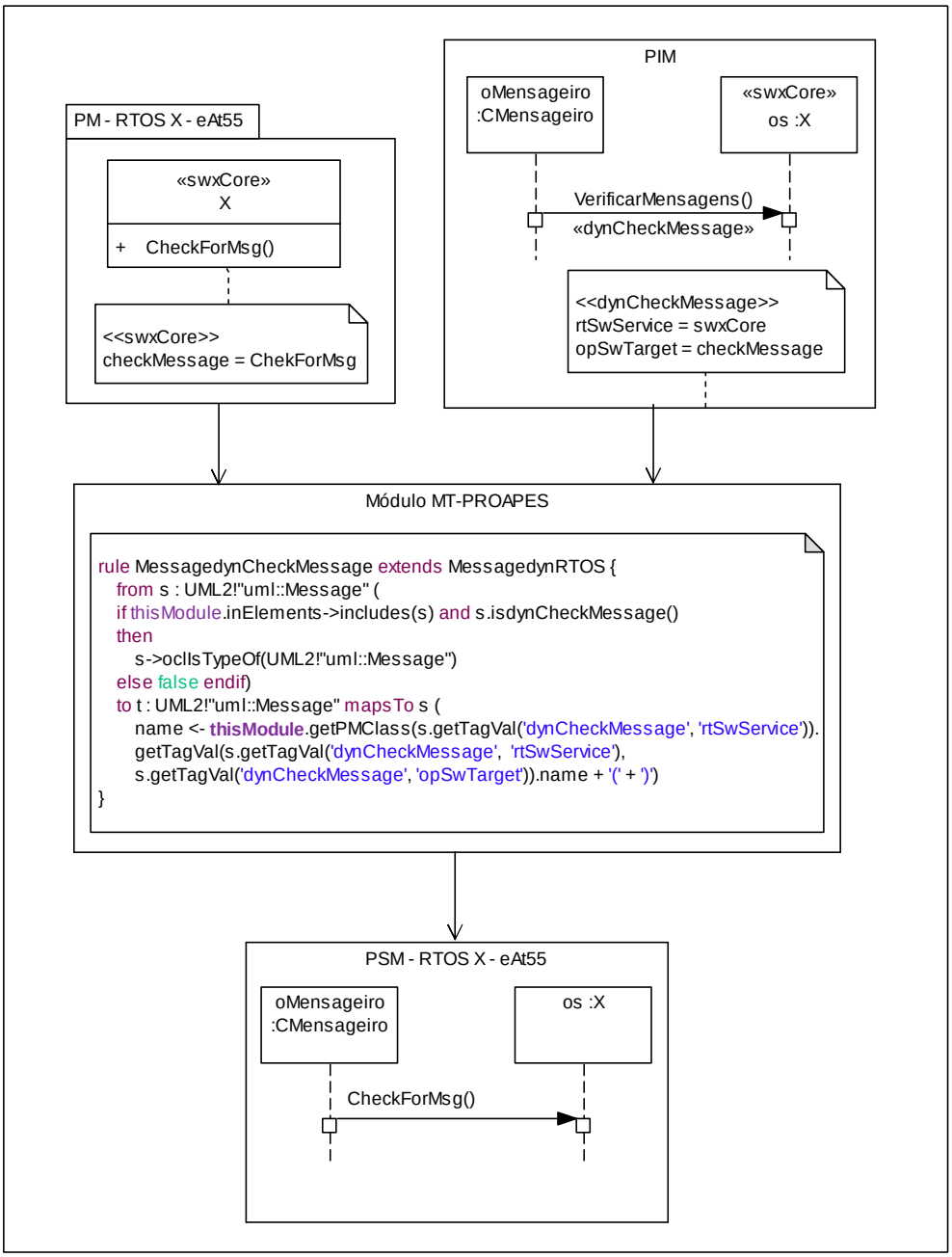

Figura 63 – Modelo de Sequência “Checar Mensagens”

5.1.6.2 Modelos de Sequência para o Pacote "Inicializar Sistema de Alarme"

A transformação de modelos do diagrama de sequência definido para o modelo de caso de uso: "Inicializar Sistema de Alarme" é ilustrada por meio de duas figuras (Figuras 64 e 65), devido a restrições de tamanho encontradas para representar todos os elementos envolvidos em uma única figura. Inicialmente, a Figura 64 ilustra o PIM dinâmico, representado por meio de um diagrama de sequência. Nesse diagrama, as mensagens são enviadas pelo objeto *oControlador* aos objetos *os* e *g_keyboard* que representam o RTOS *X Real-Time Kernel* e o teclado, respectivamente. Nessa figura, o modelo PIM possui diversas mensagens anotadas pelos estereótipos *<<dynInitRTOS>>*, *<<dynActivateThread>>*, *<<dynKeyboard>>*, *<<dynRegisterRecieverDD>>* e *<<dynInitSched>>* indicando o uso de serviços do RTOS responsáveis por inicializar o uso do RTOS, criar *threads*, inicializar o teclado, registrar o recebimento de sinais do teclado e inicializar o escalonador.

Por sua vez, a Figura 65 ilustra, de forma resumida, o PM selecionado e o módulo MT-PROAPES que indica quais as regras utilizadas para realizar a transformação do PIM em um PSM – X eAt55. Desse modo, as mensagens ilustradas no PIM são anotadas pelos estereótipos do perfil *dynSwRTOS* e *dynDDRTOS* que representam mensagens de *software* e de *hardware*, respectivamente. Essas mensagens contêm notas anexadas que ilustram o mapeamento de valores rotulados definidos nos respectivos estereótipos desses perfis para seus elementos de modelos.

O PM ilustra os serviços e atributos que são utilizados pelo módulo MT_PROAPES responsável por transformar o PIM em um PSM, de acordo com o PM RTOS X *Real-Time Kernel* e placa de *hardwar*e eAt55. Esse PM ilustra a classe *Ckeyboard* anotada pelo estereótipo *<<ddxKeyboard>>* que é uma especialização da classe *CPIO_device* anotada pelo estereótipo *<<ddxPIODevice>>*. O objetivo dessa ilustração é evidenciar que para essa plataforma a classe *CPIO_device* possui valores rotulados diferentes da plataforma *RTOS* X *Real-Time Kernel* e placa de *hardware* eLPC48, descrita na Figura 66.

A Figura 65 ainda ilustra a inserção do PIM e PM como parâmetros de entrada no módulo de transformação de modelos MT-PROAPES. As regras utilizadas são citadas de forma resumida, devido a limitações de espaço. A partir da execução do módulo MT-PROAPES é possível a geração do PSM X - eAt55.

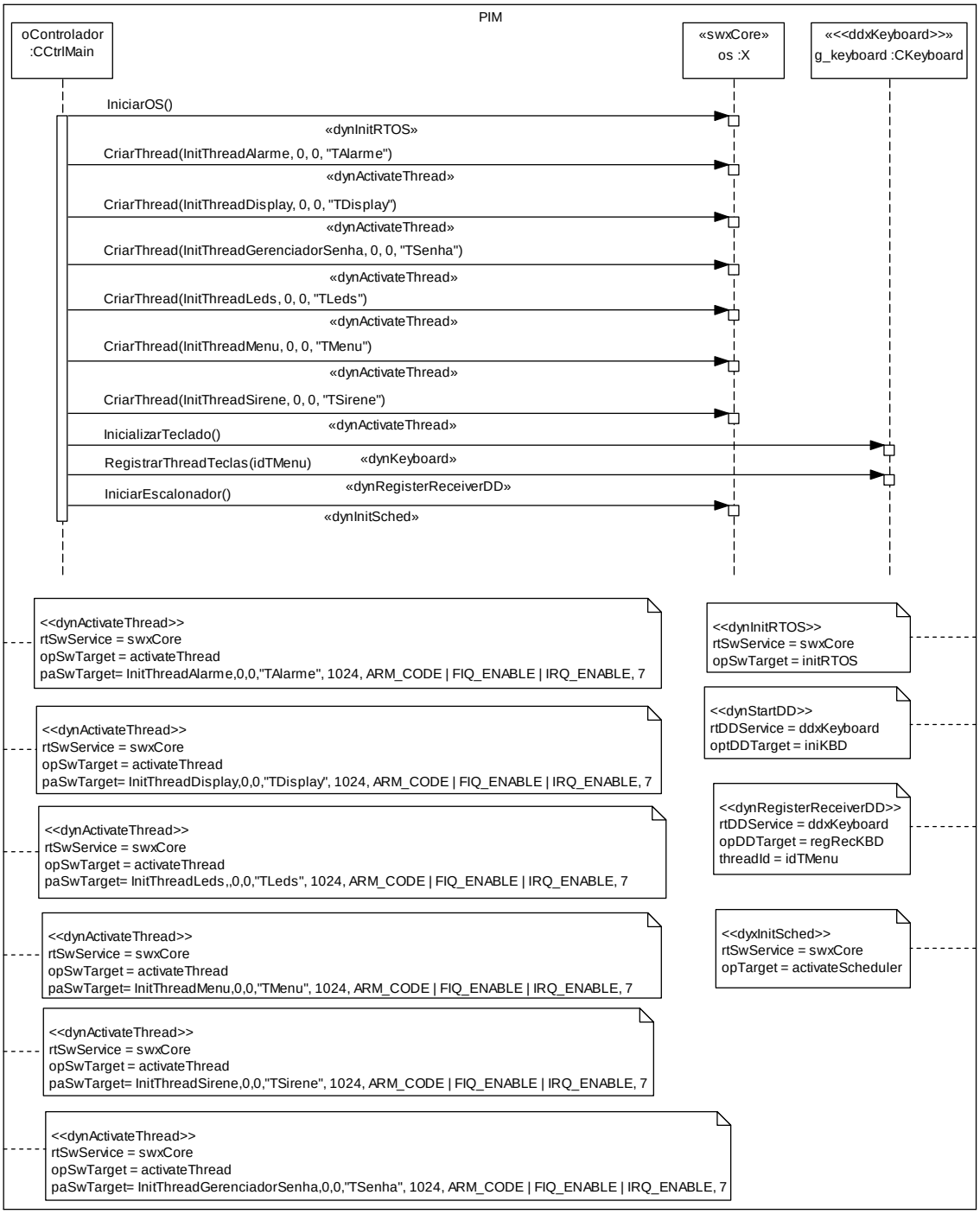

Figura 64 – Modelo de Sequência “Inicializar Sistema de Alarme”

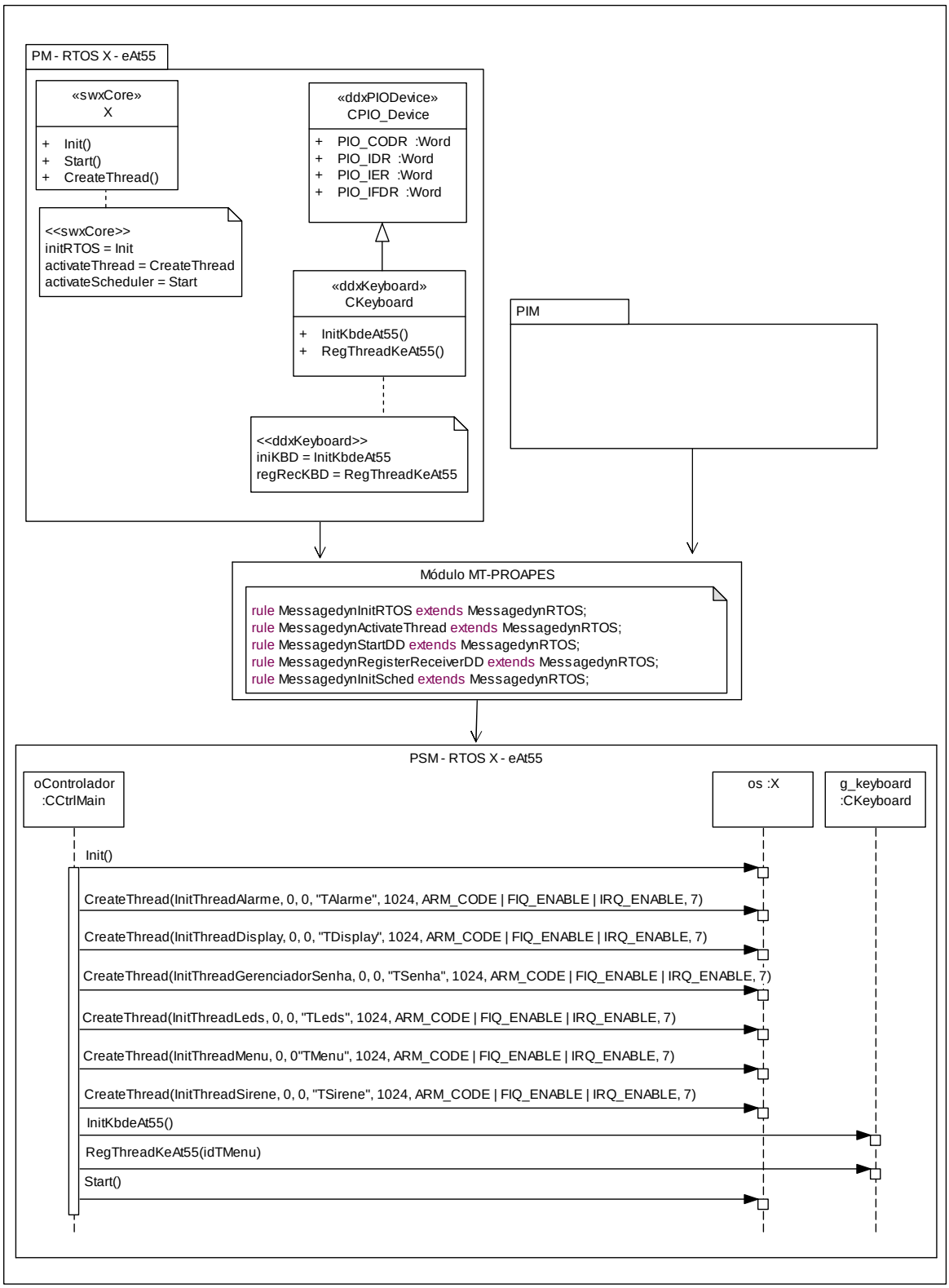

Figura 65 - Modelo de Sequência “Inicializar Sistema de Alarme” – X eAt55

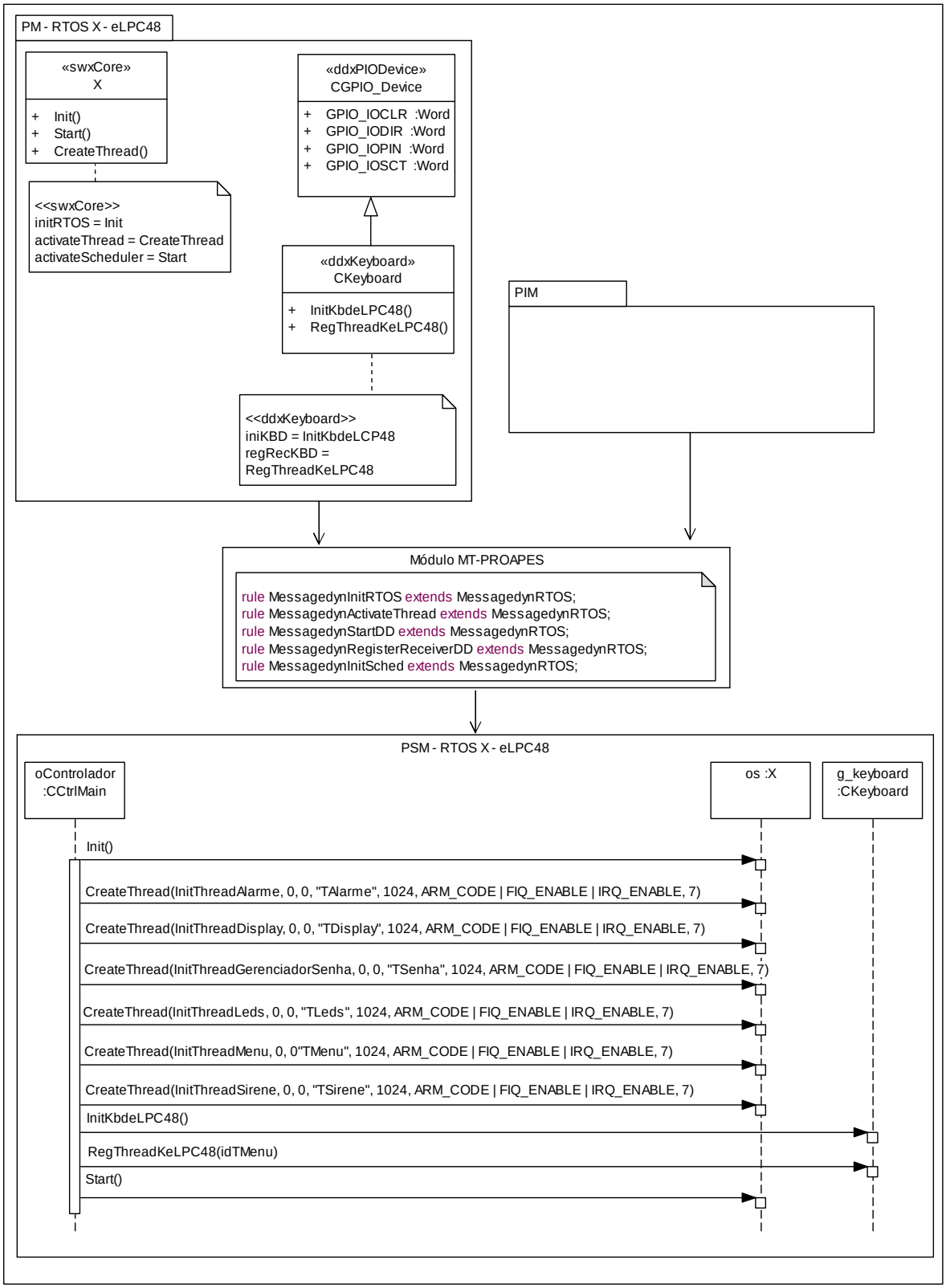

Figura 66 - Modelo de Sequência “Inicializar Sistema de Alarme” – X eLPC48

A Figura 66 apresenta a transformação do mesmo PIM (ilustrado na Figura 64) em um PSM baseado em uma plataforma diferente. Desse modo, o modelo PIM é transformado em um PSM voltado para a plataforma RTOS *X Real-Time Kernel* em placas eLPC48. As diferenças podem ser percebidas nos PMs, apresentados nas Figuras 65 e 66, que possuem atributos específicos da plataforma adotada, bem como nas operações descritas nas mensagens do PSM. O PM - X eAt55 possui uma classe denominada “CPIO_Device” anotada com o estereótipo <<*ddxPIODevice*>> e define atributos relacionados com *drivers* de dispositivo, tais como: *PIO_CODR, PIO_IDR, PIO_IER* e *PIO_IFDR*. Essa classe define os *drivers* de dispositivos *PIO* (*Peripheral Input/Output*) responsáveis em realizar configurações em periféricos de entrada e saída e define atributos para limpar o registro de saída de dados, habilitar e desabilitar o registro de dados, dentre outros. Devido a limitações de espaço, apenas alguns desses atributos são apresentados na Figura 65.

Por sua vez, no PSM - X eLPC48 (Figura 66) a classe anotada com o estereótipo <<*ddxPIODevice*>> é denominada “*CGPIO_Device*” e define diferentes atributos relacionados com *drivers* de dispositivo, tais como: *GPIO_IOCLR, GPIO_IODR, GPIO_IOPIN* e *GPIO_IOSET*. Do mesmo modo, essa classe define os *drivers* de dispositivos *PIO* (*Peripheral Input/Output*) responsáveis em realizar configurações em periféricos de entrada e saída, mas nesse caso, define os atributos específicos da plataforma eLPC48. Apenas alguns desses atributos são apresentados na Figura 66 devido a limitações de espaço.

5.1.6.3 Modelos de Sequência para o Pacote “Gerenciar LEDs”

A Figura 67 ilustra a transformação do modelo de sequência definido para o modelo de caso de uso: “Emular Alarme Ativando”. Inicialmente, a Figura 67 ilustra o *PIM*, que possui a mensagem *SuspenderTarefa* anotada pelo estereótipo <<dynSleepThreadTime>> indicando que esse serviço *RTOS* representa a suspensão momentânea da execução da *thread* LED e que *LEDS_TEMPO_ATUALIZACAO/5* refere-se ao tempo de suspensão da *thread* LED. Isso é possível devido a nota anexada ao diagrama de sequência que descreve a atribuição de valores rotulados definidos no estereótipo <<dynSleepThreadTime>> para a mensagem *SuspenderTarefa* e o parâmetro *LEDS_TEMPO_ATUALIZACAO*. O PM e o PIM utilizados como atributos de entrada do módulo MT-PROAPES também são ilustrados nesta figura.

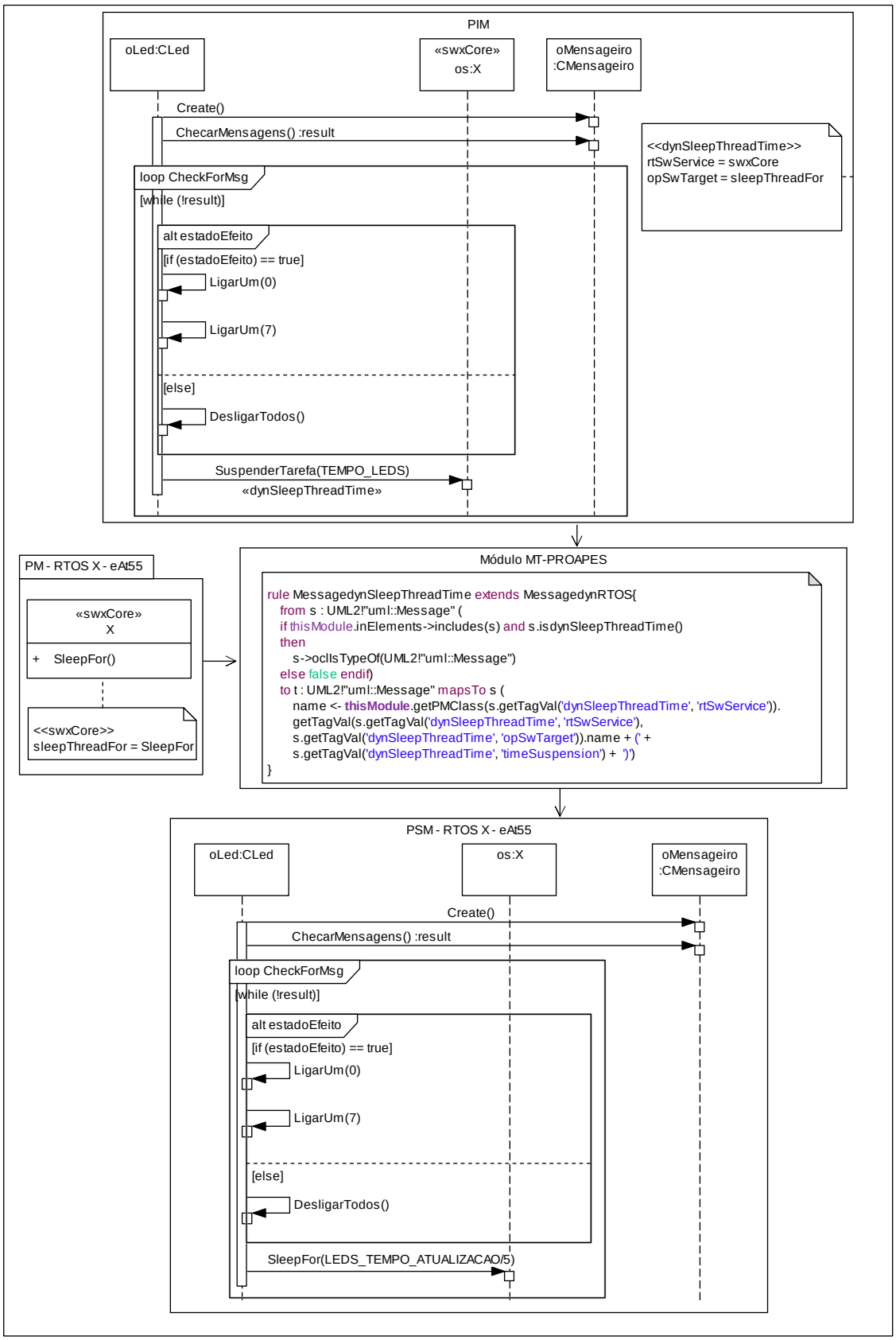

Figura 67 – Modelo de Sequência “Emular Alarme Ativando”

As Figuras 68 e 69 ilustram as transformações dos modelos de sequência definidos para os modelos de casos de uso “Emular Sirene” e “Emular Alarme Reagindo”,

respectivamente. Essas figuras ilustram características semelhantes às apresentadas na Figura 67.

5.1.6.4 Modelos de Sequência para o Pacote "Gerenciar Display"

A Figura 70 ilustra a transformação do modelo de sequência definido para o modelo de caso de uso: "Alterar Estado". Inicialmente, o PIM é apresentado e possui a mensagem *EnviarComandoDisplay* anotada pelo estereótipo <<dynSendCommandDD>> indicando que esse serviço RTOS representa o envio de um comando para o *display*. A Figura 70 também ilustra o PM selecionado e o PIM definido como atributos de entrada do módulo MT-PROAPES que define as regras usadas para transformar o PIM em um PSM - X eAt55.

As Figuras 71 e 72 ilustram as transformações dos modelos de sequência definidos para os modelos de casos de uso "Limpar *Display*" e "Mostrar Senha Incorreta", respectivamente. Essas figuras ilustram características semelhantes às características ilustradas na Figura 70. Outro modelo de casos de uso que faz parte do pacote "Gerenciar *Display*" e denominado "Mostrar Palavra" já foi apresentado na Seção 4.9.3 desta tese.

5.1.6.5 Modelos de Sequência para o Pacote "Gerenciar Sensor"

A Figura 73 ilustra a transformação do modelo de sequência definido para o modelo de caso de uso: "Gerenciar Sensor". Nessa figura, o PIM define a mensagem *SuspenderTarefa* anotada pelo estereótipo <<dynSleepThread>>, indicando que esse serviço RTOS suspende a execução da *thread* sensor momentaneamente. A Figura 73 também ilustra o PM selecionado e o PIM definido como atributos de entrada do módulo MT-PROAPES (que define as regras para realizar transformação do PIM em um PSM - X eAt55). Por limitações de espaço, apenas uma regra do módulo MT-PROAPES é apresentada nessa figura.

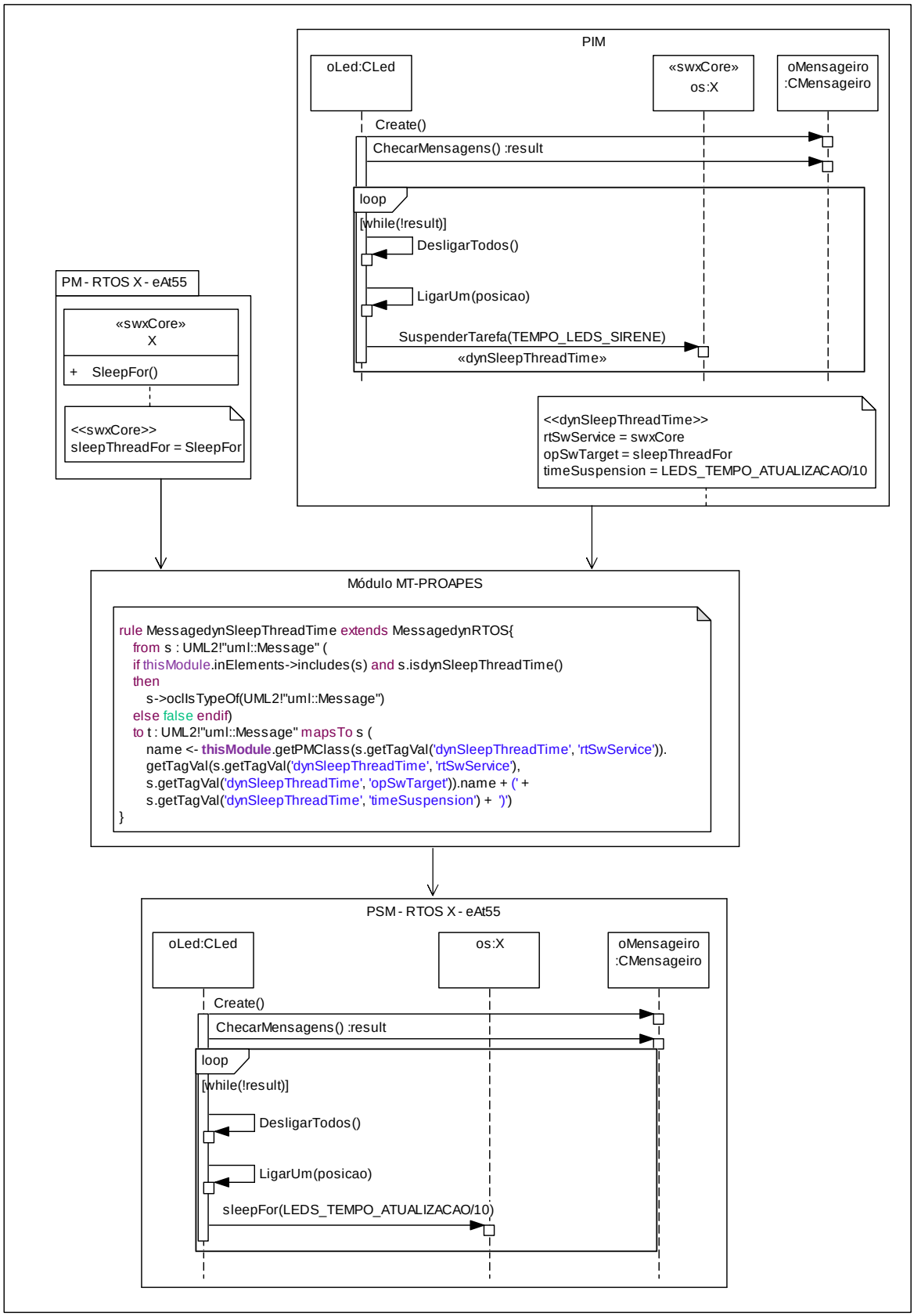

Figura 68 – Modelo de Sequência “Emular Sirene”

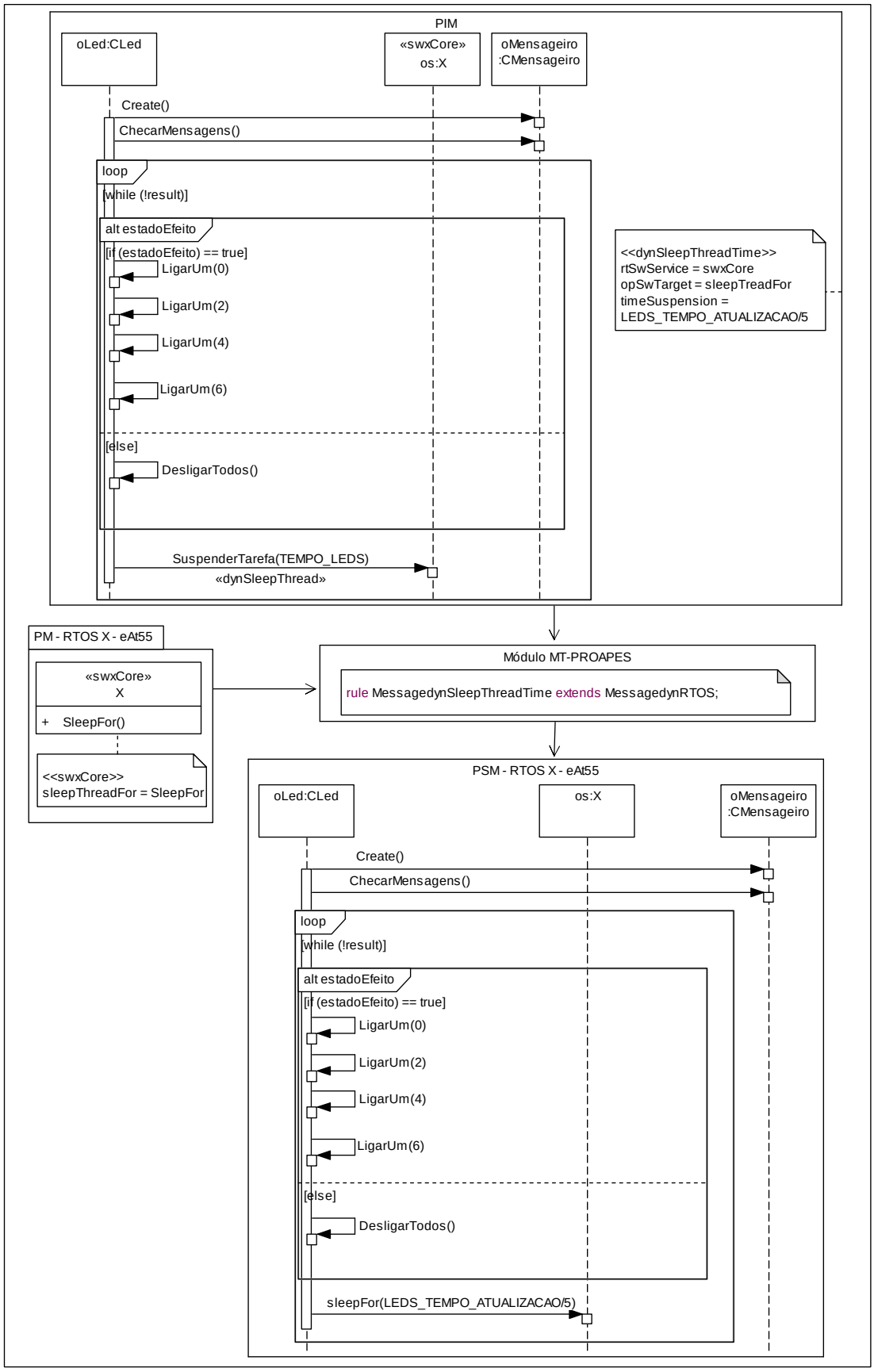

Figura 69 – Modelo de Sequência “Emular Alarme Reagindo”

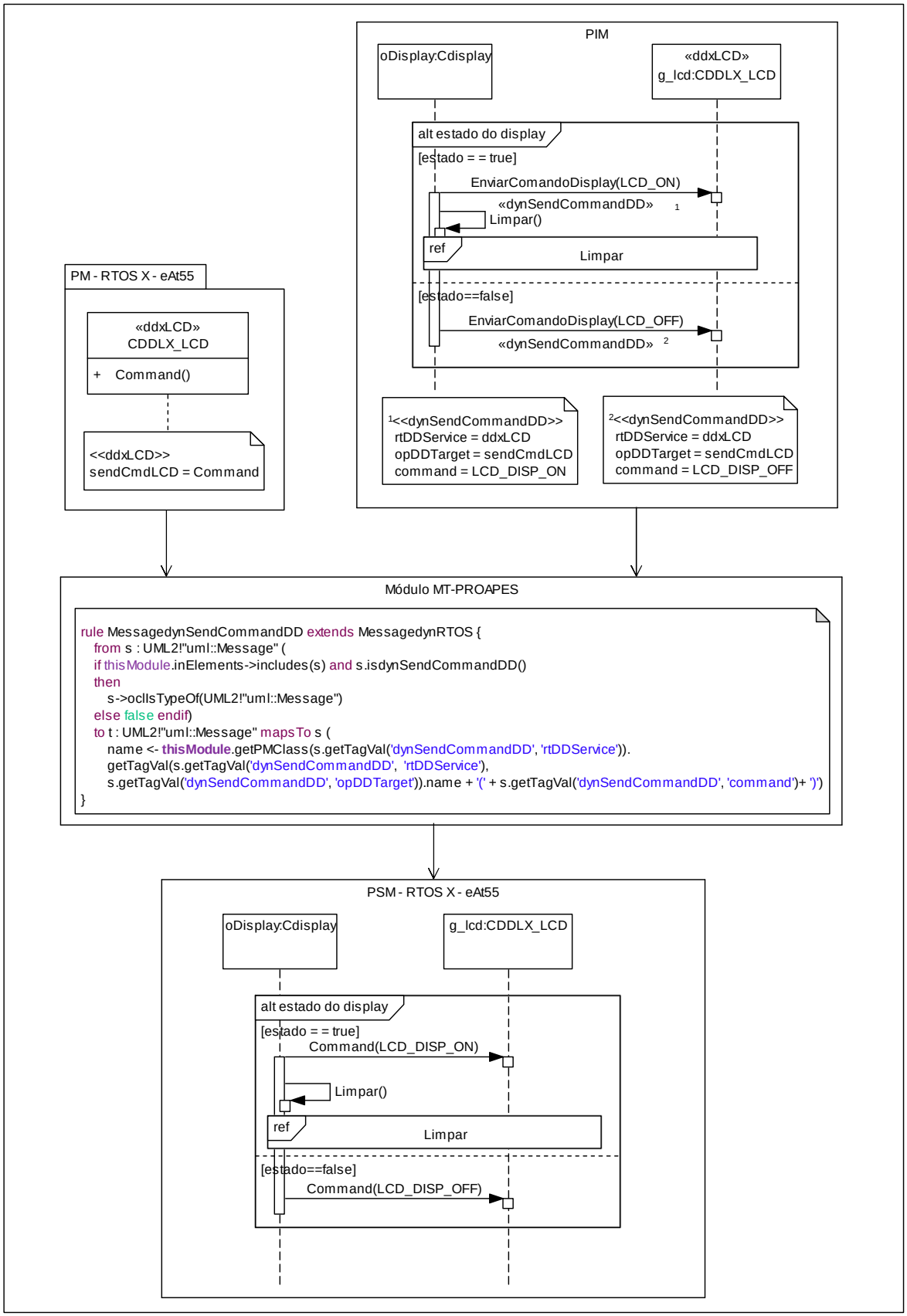

Figura 70 – Modelo de Sequência “Alterar Estado”

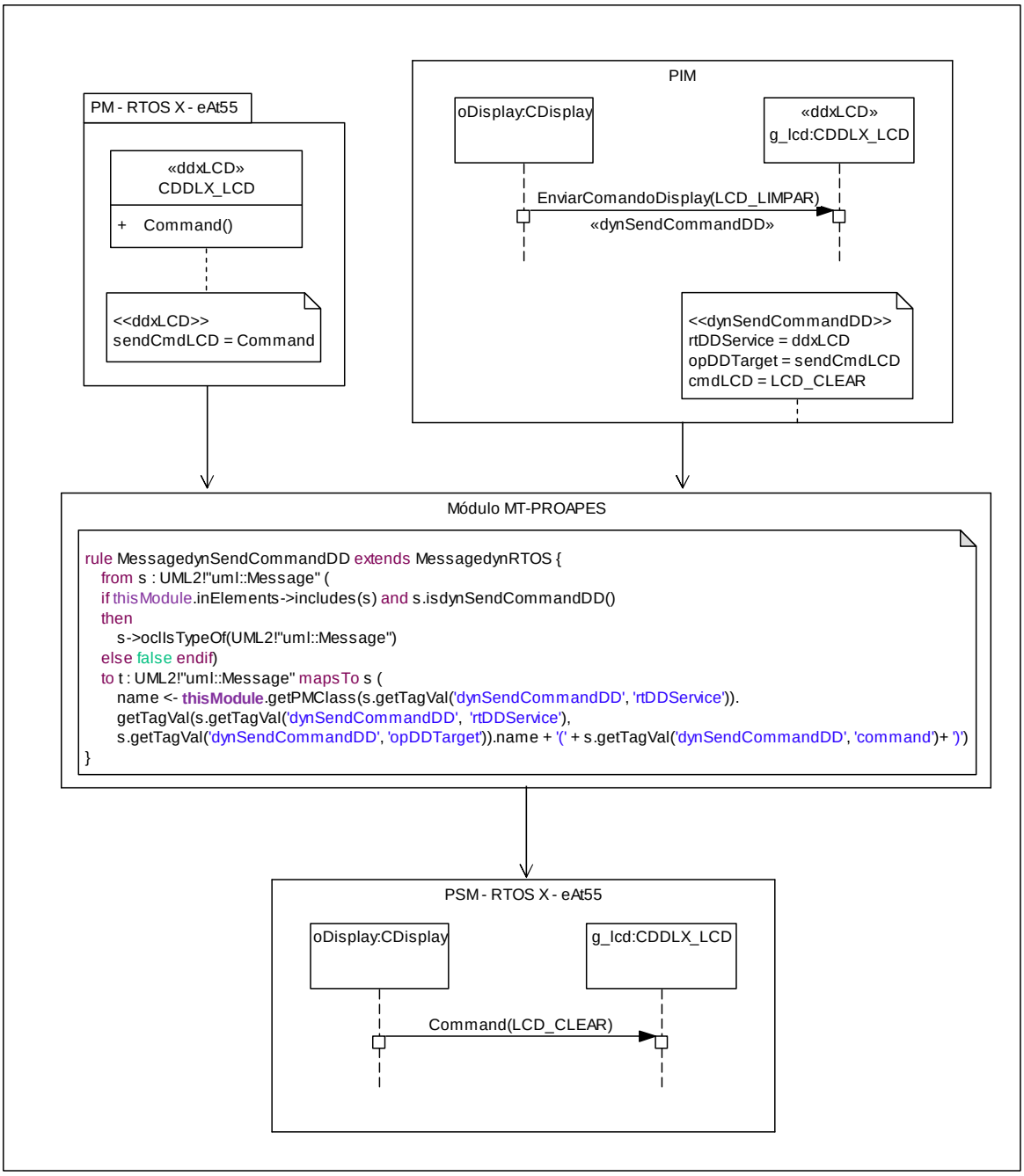

Figura 71 – Modelo de Sequência "Limpar *Display*"

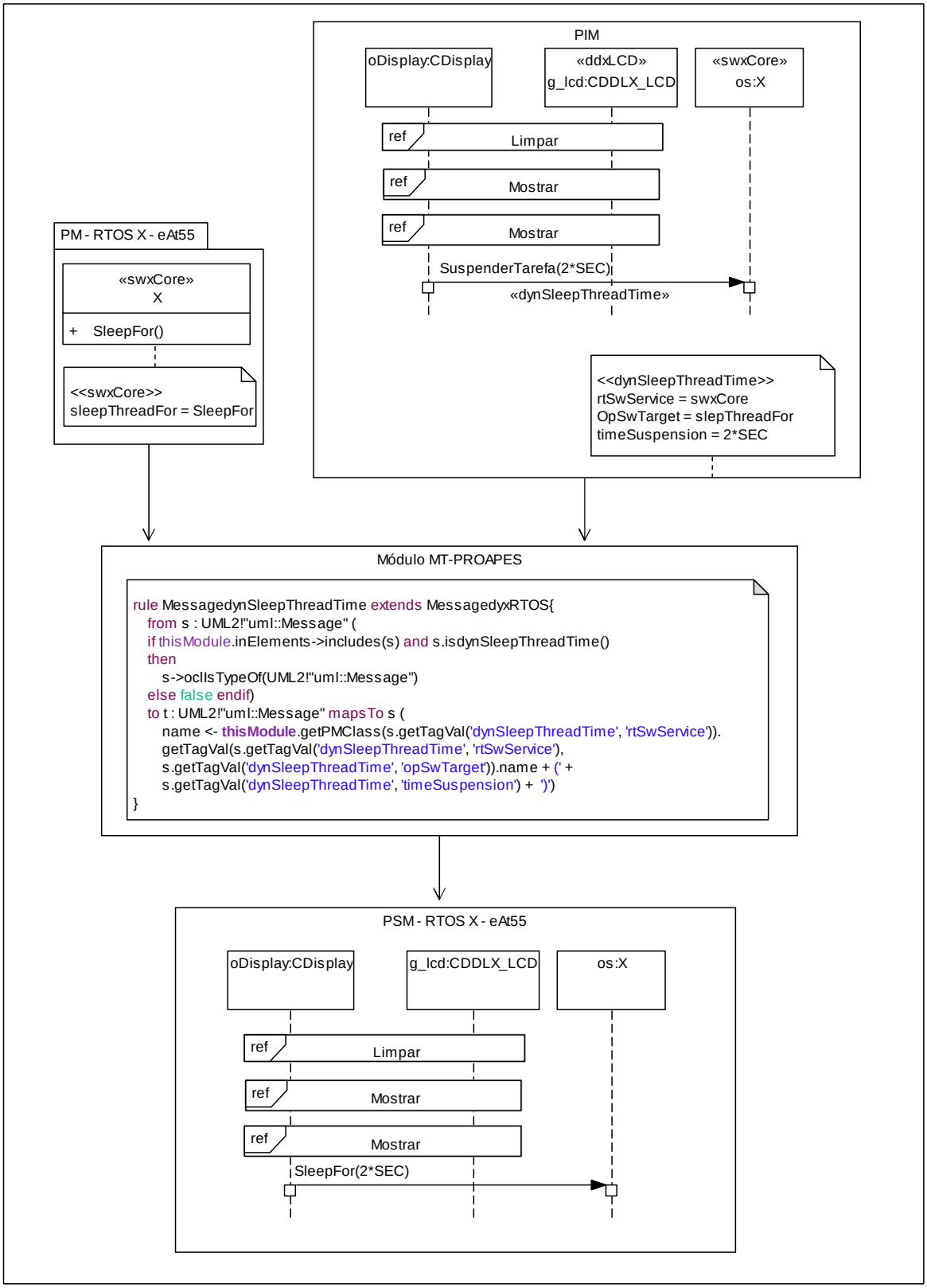

Figura 72 – Modelo de Sequência “Mostrar Senha Incorreta”

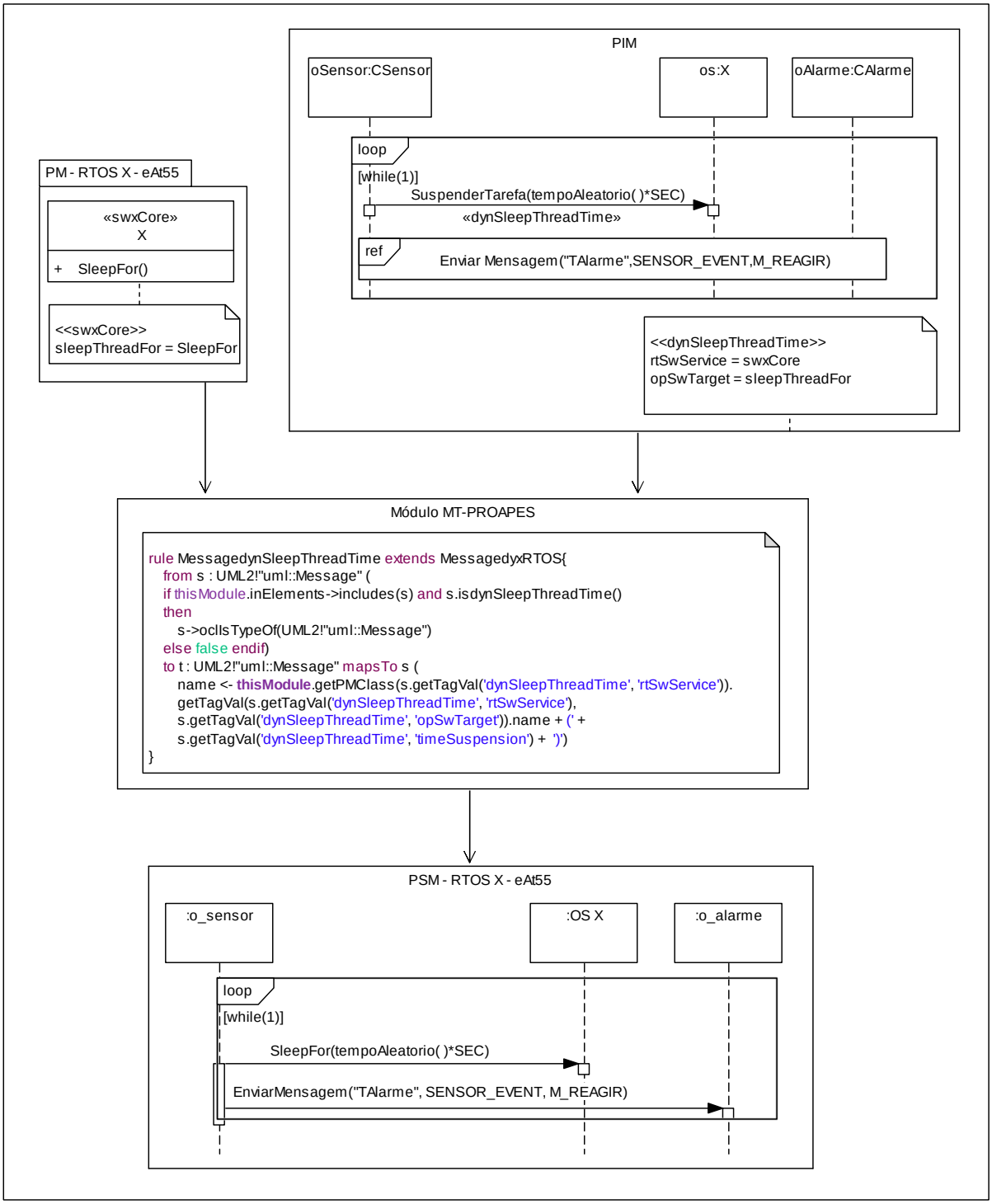

Figura 73 - Modelo de Sequência “Gerenciar Sensor”

5.1.6.6 Modelo de Classes do SA

A Figura 74 ilustra o modelo PIM para o sistema SA, assim, um diagrama de classes estático é criado. Nesse modelo são aplicados os estereótipos <<*rtSwOperation*>> e <<*rtDDOperation*>>, definidos no perfil de modelagem de aplicação (*Aplication Modeling Profile* - AMP). Esses estereótipos indicam a ocorrência de serviços de RTOS de forma genérica, pois os mesmos estão relacionados com o perfil AMP, um dos perfis que compõem o perfil PROAPES, e não com um PM específico. Desse modo, o engenheiro de *software* precisa conhecer que existem serviços *RTOS* sendo usados e onde precisam ser usados, mas não precisa conhecer detalhes de uma plataforma específica.

Dado que, a ocorrência dos estereótipos <<*rtSwOperation*>> e <<*rtDDOperation*>> acontece mais de uma vez no diagrama apresentado na Figura 74, ou seja, cada um desses estereótipos é aplicado em mais de uma operação nesse diagrama, foram inseridos números juntamente com os estereótipos aplicados, visando identificar cada inserção de estereótipo nesse modelo. Desse modo, é possível identificar a qual nota do diagrama essa aplicação do estereótipo se refere, em específico.

Nesse exemplo, o módulo MT-AMP é responsável por realizar a transformação do modelo PIM em um PSM.

As principais regras que compõem o módulo MT-AMP e que são utilizadas nessa transformação são ilustradas na Figura 75. A regra “OperationRtSw” localiza as operações do modelo *PIM* anotadas com o estereótipo <<rtSwOperation>> e, com base nas informações das propriedades relacionadas com esse esterótipo, realiza ações específicas para inserir características de plataforma referentes a serviços de *software* de RTOS no PSM.

Por sua vez, a regra “OperationRtDD” busca as operações do modelo-fonte marcadas com o estereótipo <<rtDDOperationt> e, com base nas informações das propriedades desse esterótipo, realiza ações específicas para inserir características de plataforma referentes a serviços de device drivers de RTOS no PSM. Por fim, a regra “*Model*” copia as propriedades do modelo PIM para o modelo PSM, bem como insere os novos elementos criados pelas regras “OperationRtSw” e “OperationRtDD” no modelo-alvo (PSM).

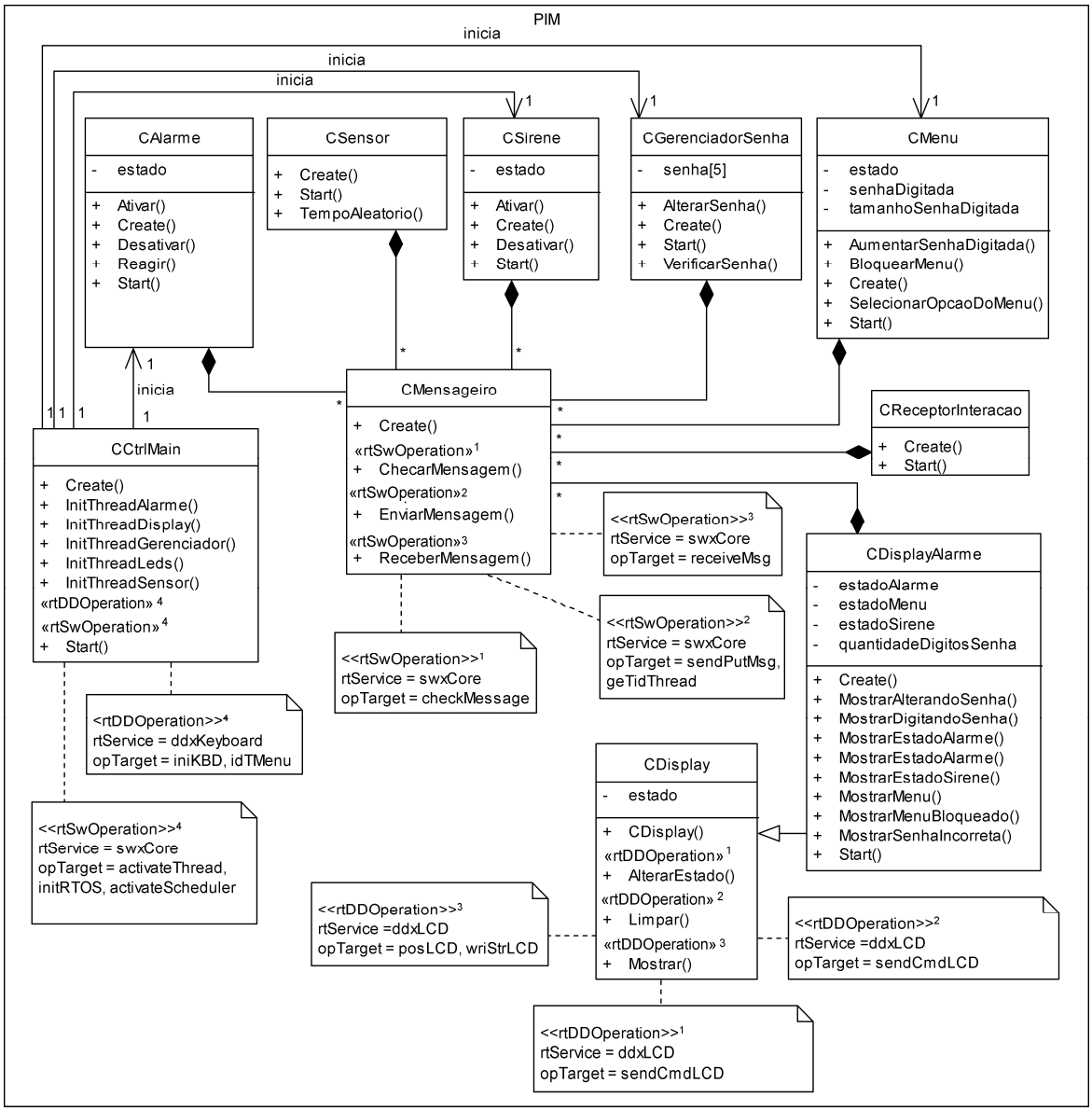

Figura 74 – Diagrama de Classes do SA

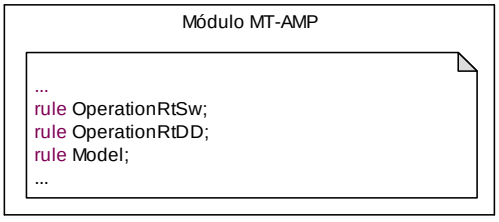

Figura 75 – Principais regras do Módulo MT-AMP utilizadas

Os PSMs gerados pela execução da transformação para as placas eAt55 e eLPC48 são ilustrados nas Figuras 76 e 77, respectivamente. O PSM - X eAt55, ilustrado na Figura 76, apresenta as classes existentes no PIM (Figura 74) e as classes adicionais que são classes descritas no PM para a plataforma RTOS *X Real-Time Kernel* e placa eAt55. As classes adicionais são a classe “X” que possui serviços RTOS e as classes de *drivers* de dispositivos *CKeyboard* e *CDDX_LCD* (especializações da classe *CPIO_Device* responsável em realizar configurações nesses *drivers* de dispositivos).

Por sua vez, o PSM - X eLPC48 ilustrado na Figura 77 ilustra as classes existentes no PIM (Figura 77), porém, ilustra classes adicionais descritas no PM para a plataforma RTOS *X Real-Time Kernel* e placa eLPC48. A classe “X” que possui serviços RTOS e as classes de *drivers* de dispositivos *CKeyboard* e *CDDX_LCD*, essas duas classes são especializações da classe *GPIO_Device* responsável em realizar configurações nesses *drivers* de dispositivos.

Observa-se que, as diferenças dos PSMs gerados nesse exemplo estão ilustradas nas classes *CKeyboard* e *CDDX_LCD*. Na Figura 76, o PSM - X eAT55, possui a classe *Ckeyboard* que contêm os métodos: *initKbdeAt55* (responsável em inicializar o teclado) e *RegThreadKeAt55* (responsável em gerenciar a *thread* do teclado). Além disso, a classe *Ckeyboard* é uma especialização da classe *CPIO_Device* que possui configurações próprias para os *drivers* de dispositivos de entrada e saída de processadores *Atmel*.

Por sua vez, na Figura 77 o PSM - X eLPC48, possui a classe *Ckeyboard* que contém os métodos: *initKbdeLPC48* (responsável em inicializar o teclado) e *RegThreadKeLPC48* (responsável em gerenciar a *thread* do teclado). Além disso, a classe *Ckeyboard* é uma especialização da classe *GPIO_Device* que possui configurações próprias para os *drivers* de dispositivos de entrada e saída de processadores NXP.

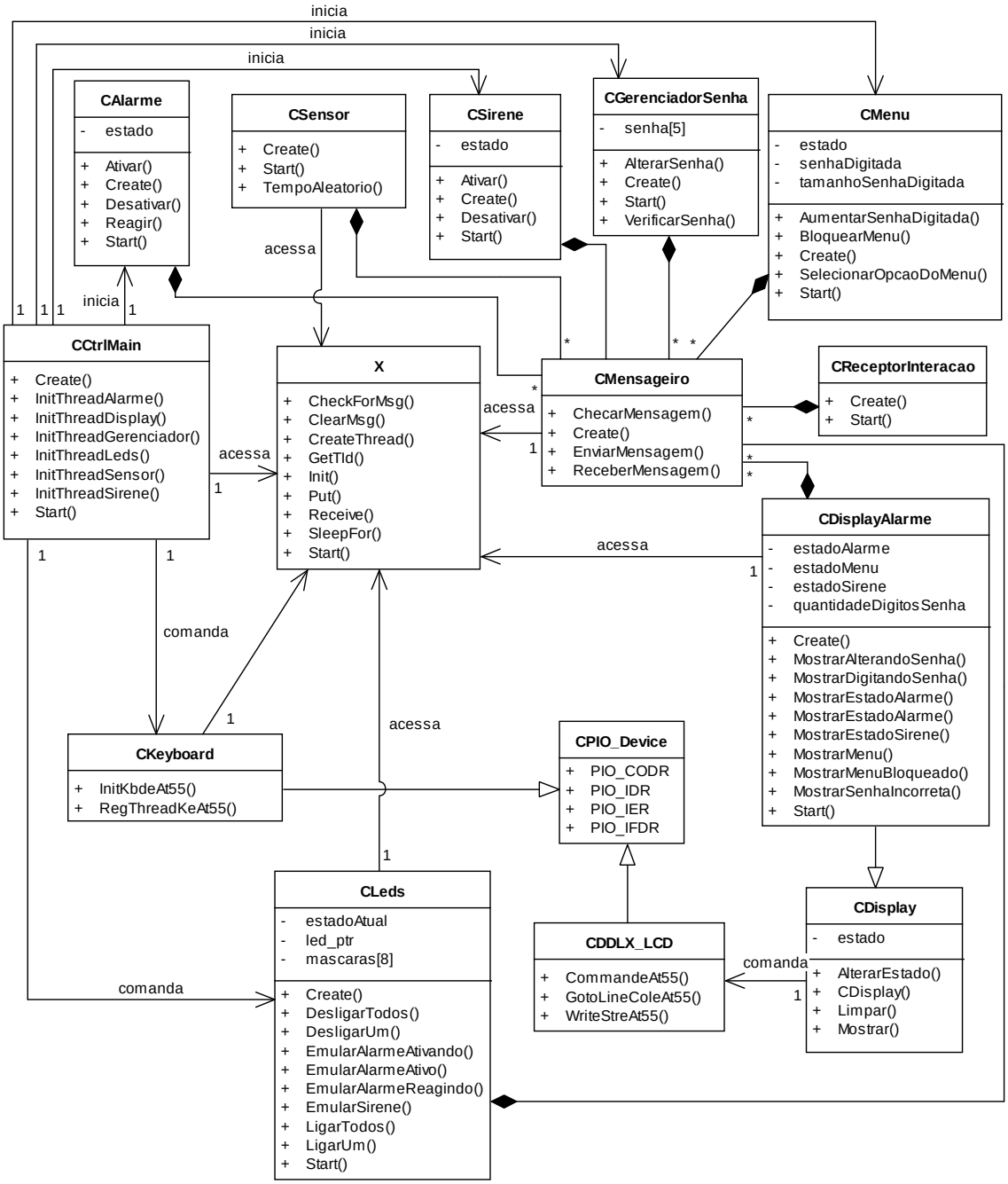

Figura 76 - Diagrama de classes referente ao PSM X – eAt55

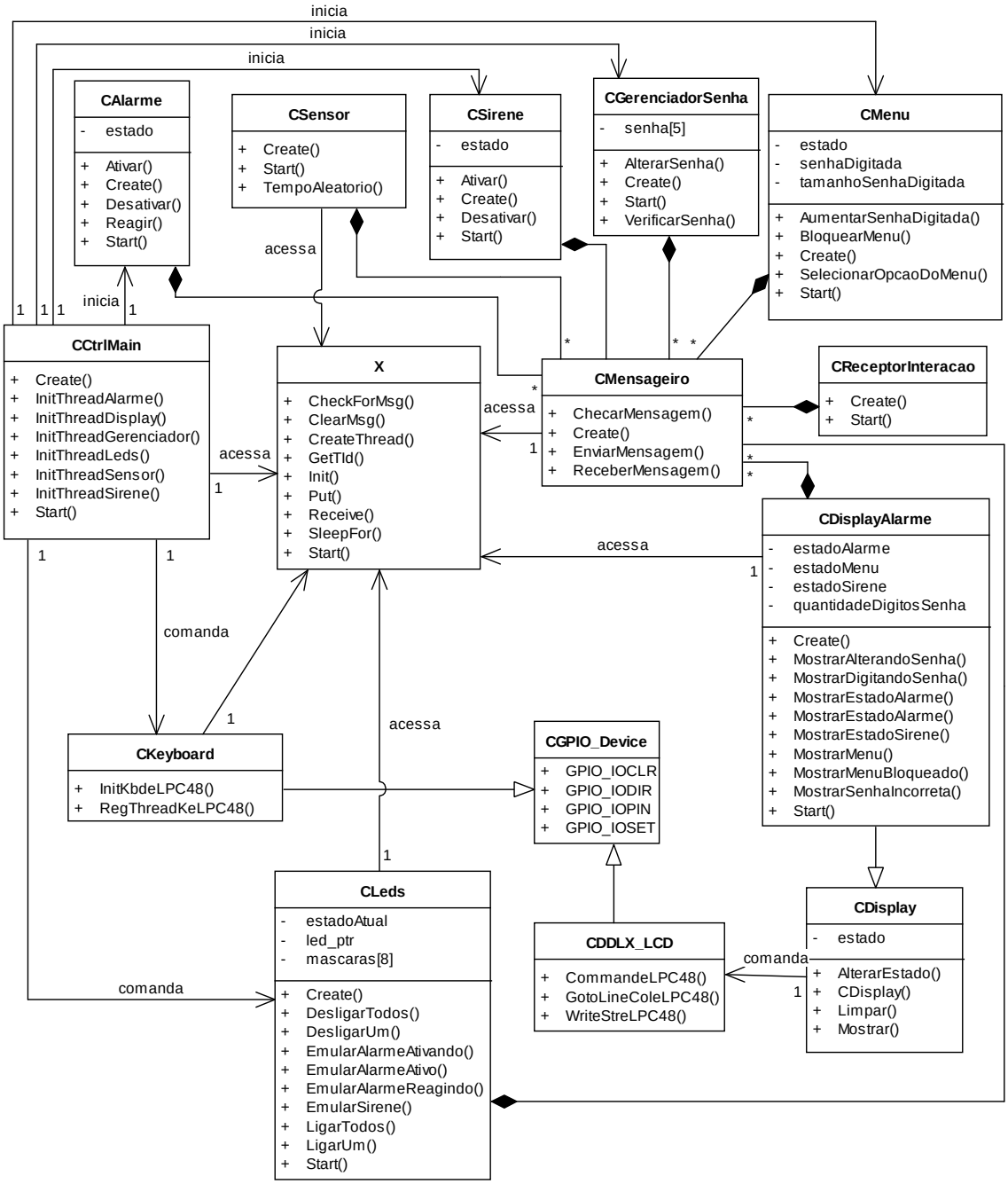

Figura 77 - Diagrama de classes referente ao PSM X – eLPC48

5.1.7 Discussões

Com base no caso de estudo desenvolvido foi possível realizar uma comparação qualitativa da aplicação do método PI-MT (baseado na abordagem MDA) com relação à

aplicação da abordagem tradicional de desenvolvimento de *software*. Com relação ao uso do PI-MT, é preciso considerar os custos envolvidos com o aprendizado, por parte do desenvolvedor da aplicação, relacionado com os estereótipos e propriedades do perfil de ligação que devem ser aplicados no PIM. Outro custo refere-se à definição das regras de transformação PIM-para-PSM. É relevante ressaltar que em projetos futuros este custo é absorvido e, desse modo, não mais considerado.

O uso do PI-MT exigiu um menor conhecimento, por parte do desenvolvedor do PIM, com relação à plataforma de implementação. Na abordagem MDA, as características relacionadas com a plataforma-alvo não foram definidas no modelo da aplicação, visto que o PIM foi modelado e, posteriormente, transformado de forma automatizada em um PSM. Por sua vez, na abordagem tradicional a especificação dos detalhes específicos da plataforma-alvo foi mandatória. Desse modo, é possível concluir que, em projetos iniciais o uso do PI-MT tem um custo inicial de treinamento e de definição das regras de transformação, entretanto, em projetos futuros benefícios significativos são obtidos, facilitando a construção do sistema, bem como, possibilitando o reuso da transformação de modelos para diferentes plataformas.

5.2 CONCLUSÕES DO CAPÍTULO

Uma questão crucial a ser considerada no desenvolvimento de *software* embarcado baseado em RTOS utilizando a abordagem MDA reside na separação entre os interesses relacionados à plataforma alvo e à transformação de modelos. Desse modo, por meio do caso de estudo do sistema SA foi possível apresentar um exemplo real da separação entre esses interesses.

Por meio deste caso de estudo foi possível ilustrar a transformação de modelos realizada em plataformas diferentes, ou seja, o sistema projetado nesse caso de estudo foi desenvolvido para as plataformas: RTOS *X Real-Time Kernel* e placa de avaliação eSystech eAt55 e RTOS *X Real-Time Kernel* e placa de avaliação eSystech eLPC48. Algumas diferenças foram evidenciadas em modelos do caso de estudo do SA.

O uso das transformações de modelos MT-PROAPES e MT-AMP foi exemplificado por meio do caso de estudo apresentado, com base em modelos de sequência e modelos de classes. Desse modo, foi possível obter percepções práticas dos aspectos técnicos ao desenvolver o sistema SA utilizando duas abordagens distintas de desenvolvimento para o

RTOS *X Real-Time Kernel* e duas plataformas de *hardware* distintas: placa eAt55 e placa eLPC48.

O mesmo sistema (SA) foi implementado utilizando-se a abordagem de desenvolvimento de *software* tradicional e a abordagem de desenvolvimento dirigida a modelos. Em um primeiro momento, pode-se dizer que o desenvolvimento de *software* dirigido a modelos é mais complexo e difícil de ser realizado. Isto ocorre devido à necessidade de criação do programa (módulo) de transformação de modelos. Porém, em um segundo momento, i.e., em implementações futuras que possam utilizar as regras de transformação de modelos já definidas, o processo de desenvolvimento torna-se mais produtivo e a portabilidade é melhorada.

Esta tese tem como foco específico a etapa de transformação intermediária no ciclo de desenvolvimento MDA, i.e., a transformação de um PIM em um PSM. Portanto, o tema da transformação do PSM em código ultrapassa o escopo desta tese.

Quando o ciclo de transformações estiver completo será possível reduzir a realização de tarefas redundantes (modelo e código do *software*), pois o código será gerado automaticamente (ou semi-automaticamente) a partir dos modelos PIMs. Em adição, a sincronização entre modelos e código também será facilitada, por meio de sucessivas transformações (1 - PIM-para-PSM e 2 - PSM-para-código). É pertinente salientar que a primeira etapa de transformação do ciclo da MDA (PIM-para-PSM) é considerada a mais complexa, pois em transformações PSM-para-código, as informações relativas à plataforma já se encontram inseridas no modelo-fonte (KLEPPE *et al*., 2003).

6 CONCLUSÕES, RESULTADOS E TRABALHOS FUTUROS

Este capítulo apresenta as conclusões deste trabalho de pesquisa, trazendo as discussões a respeito da sua realização e a descrição dos resultados. Por fim, os possíveis trabalhos futuros são apresentados.

6.1 CONCLUSÕES DO TRABALHO

O desenvolvimento desta tese exigiu grande esforço de pesquisa visando consolidar os conhecimentos na área de Engenharia de *Software*, bem como buscar novos conhecimentos na área de abordagens dirigidas a modelos. O estado da arte e o estado da técnica nessas áreas foram levantados. O conhecimento na área de sistemas embarcados e conceitos relacionados também foram aprimorados. Todas essas pesquisas serviram como fundamentação teórica necessária para se atingir o objetivo esperado.

A avaliação da autora desse trabalho de pesquisa é bastante positiva, dado que a pesquisa realizada é relevante e desafiadora, inserida no contexto da Arquitetura Dirigida a Modelos (MDA) que é uma das principais tendências da área de Engenharia de *Software*. Muitos avanços têm sido realizados na área de MDA, entretanto, existem ainda importantes questões a serem resolvidas. Uma dessas questões está relacionada com a transformação de modelos que possui um papel fundamental no processo de desenvolvimento dirigido a modelos. Entretanto, existe uma grande lacuna em termos de métodos e de ferramentas para guiar e auxiliar o desenvolvimento de transformação de modelos em MDA (BAUDRY *et al*., 2010; FRANCE e RUMPE, 2007; ALMEIDA, 2006). Esta tese contribuiu para preencher esta lacuna, propondo um método para criação de transformações de modelos no qual são estabelecidos os passos a serem realizados e elementos de entrada e saída bem definidos.

A realização desta tese foi desafiadora, principalmente por concentrar-se em sistemas embarcados. O desenvolvimento de sistemas embarcados exige uma abordagem sistemática, bem como suporte de ferramentas avançadas para lidar com a grande diversidade de plataformas existentes e com a crescente complexidade desse tipo de sistema. Em adição, o mercado de *software* embarcado apresenta um grande potencial de expansão, necessitando, dessa forma, métodos, técnicas e ferramentas que aumentem os níveis de qualidade e produtividade no processo de desenvolvimento de *software* embarcado. A MDA apresenta-se

como uma das alternativas para aumentar a qualidade e incrementar a produtividade no processo de desenvolvimento de *software* (FRANCE e RUMPE, 2007; KLEPPE *et al*., 2003).

Para aumentar o potencial de uso de ferramentas MDA voltadas para o desenvolvimento de *software* embarcado, na prática, é importante que a técnica de transformação de modelos utilizada seja minimamente dependente da especificação das características da plataforma. Além disso, reduzindo a dependência da transformação de modelos com relação aos aspectos das plataformas, o impacto causado pela grande diversidade de plataformas existentes, no contexto de sistemas embarcados, é reduzido.

Finalmente, o desenvolvimento de sistemas embarcados baseados em Sistemas Operacionais em Tempo-Real (RTOS) é uma área de pesquisa pouco explorada no contexto da MDA (JEON *et al*., 2009; KARSAI *et al*., 2006). Desse modo, esta tese de doutoramento se destaca por abordar justamente os temas supracitados: sistemas embarcados baseados em RTOS, contribuindo assim, com soluções para o desenvolvimento de *software* nessa área.

O objetivo da tese foi cumprido por meio da proposição do método PI-MT (*Platform Independent - Model Transformations*) que define os elementos necessários para criar transformações de modelos genéricas (não dedicadas) em ambientes embarcados e baseados em RTOS. Pela aplicação do casode estudo e pelas comparações, viu-se que o método PI-MT é viável e, realmente, pode colaborar no desenvolvimento MDA de sistemas embarcados baseados em RTOS. Ainda, esse método é viável de ser integrado em ferramentas de desenvolvimento de *software* embarcado que utilizam a abordagem MDA, pois está fundamentado em padrões amplamente adotados e difundidos na literatura, como MOF, UML e ATL.

O método tem como base a metamodelagem, fundamentada pelo MOF. A vantagem que isso traz é que o MOF é apoiado por um número crescente de ferramentas e implementações de código aberto disponíveis. Por sua vez, a UML é considerada a linguagem de modelagem padrão para o desenvolvimento de *software* orientado a modelos. Ainda, atualmente, a ATL é a linguagem mais conhecida e adotada no desenvolvimento de transformações de modelos. O uso de tecnologias padronizadas possibilita uma maior disseminação da MDA na indústria de *software*.

Os principais benefícios obtidos pelo uso do método proposto na criação de transformações de modelos são a melhoria da produtividade e da portabilidade no desenvolvimento de *software* embarcado. Esses benefícios são obtidos por meio da "independência de plataforma" que possibilita um maior reuso dos modelos de *software* e das transformações de modelos. Além disso, o método oferece diretrizes para uma importante

etapa do ciclo de vida de desenvolvimento MDA, a transformação de um modelo PIM em um modelo PSM. Essa etapa antecede a etapa responsável por transformar o PSM em código-fonte do sistema de *software*. Assim, o método também contribui para o problema da geração de código, fornecendo subsídios para resolver uma etapa intermediária de transformação.

O método PI-MT é baseado na abordagem MDA e, portanto, adota as tecnologias sugeridas pelo OMG. Entretanto, esse método pode ser facilmente adaptado para outras abordagens dirigidas a modelos, como MDE e MDD. Ainda, as transformações criadas com base no PI-MT utilizaram o perfil PROAPES. Esses perfil foi desenvolvido dentro do mesmo grupo de pesquisa da UTFPR e é voltado para a definição de elementos e serviços do RTOS X *Real-Time Kernel*. O perfil PROAPES foi utilizado por possuir elementos bem definidos e por oferecer uma interface amigável, de fácil entendimento para o desenvolvedor. Entretanto, o método proposto é genérico, i.e., não está vinculado a um perfil de plataforma e a um perfil de ligação específico. Desse modo, novas transformações criadas com base no PI-MT podem utilizar outros perfis como base para sua implementação.

Outro perfil que possibilita a modelagem de sistemas embarcados baseados em RTOS é o *Modeling and Analysis of Real-Time and Embedded Systems* (MARTE). Entretanto, a ampla generalidade do MARTE torna o seu uso complexo (SILVESTRE, 2012). Por sua vez, o perfil PROAPES, utilizado como perfil de plataforma nesta pesquisa de doutorado, tem como foco a modelagem de uma família de plataformas similares. Desse modo, o PROAPES é mais fácil de ser compreendido e utilizado.

6.2 DISCUSSÕES

As discussões apresentadas nesta subseção referem-se aos desafios encontrados e relacionados com as implementações MT-AMP e MT-PROAPES realizadas nesta tese. Inicialmente, a ferramenta *Enterprise Architect* (EA) foi adquirida e estudada, na tentativa de viabilizar a implementação das transformações propostas por meio dessa ferramenta (SPARK SYSTEMS, 2012). A *Enterprise Architect* é uma ferramenta consolidada no mercado de modelagem UML e oferece recursos como boa interface e facilidade de uso. Seguindo a tendência das abordagens dirigidas a modelo, esta ferramenta também oferece recursos para o desenvolvimento dirigido a modelos, possibilitando, inclusive, gerar e executar transformações de modelos nesse ambiente.

Entretanto, foi possível verificar que a EA possui um escopo bastante limitado no que se refere às transformações e que atende a um grupo de plataformas específicas definidas por meio de pacotes, dentre elas: C#, Java e JUnit. Também, as regras de transformação são previamente definidas em *templates* e, portanto, limitadas a atender a um conjunto de problemas comuns. As principais dificuldades encontradas relacionam-se com o uso de múltiplos modelos de entrada e com a recuperação de valores de propriedades vinculados com estereótipos aplicados em elementos do modelo-fonte. Desse modo, devido às limitações apresentadas pela EA, não foi possível utilizá-la na implementação das transformações MT-AMP e MT-PROAPES. A busca por uma MTL (*Model Transformation Language*) e por um ambiente continuou e culminou na escolha da linguagem ATL e do ambiente TopCased, respectivamente.

A ATL foi selecionada como a linguagem de transformações de modelos a ser utilizada nesta tese por possuir os recursos necessários para a criação de transformações de modelos que envolvem mais de um modelo de entrada, além de possibilitar o uso de construtores imperativos nos casos em que construtores declarativos não conseguem resolver o problema. Entretanto, a ATL ainda possui algumas importantes limitações relacionadas com a rastreabilidade, complexidade de uso, pouca documentação e refinamento de modelos (JOUAULT, 2008; AGNERb *et al*., 2012; TROYA e VALLECILLO, 2011). Ainda, segundo Vara *et al*. (2009) e Cortellessa *et al*. (2008), a ATL é uma linguagem de transformação de modelos relativamente nova e que apresenta importantes desafios e limitações a serem explorados.

A ATL está incorporada ao TopCased, um ambiente voltado para o desenvolvimento dirigido a modelos, que além da ATL, oferece ferramentas de modelagem UML e ferramentas OCL, integradas em um ambiente único. Este ambiente é baseado no Eclipse e utilizado em importantes projetos de *software*, como os citados em (CRÉGUT *et al*., 2010; BERTHOMIEU *et al*., 2008; PONTISSO e CHEMOUIL, 2006).

A escolha da técnica Superposição de Módulos para realizar o refinamento de modelos UML foi bastante positiva, visto que a técnica do *refining* oferecida pela linguagem ATL não suporta os recursos exigidos para realizar as transformações propostas (AGNERb *et al*., 2012).

6.3 RESULTADOS

O método PI-MT proposto nesta tese de doutoramento define e detalha os passos a serem realizados para criação e uso de transformações de modelos reutilizáveis baseadas em RTOS. Os elementos utilizados como entrada e saída do processo de transformação também foram identificados e detalhados. Também, um padrão para o Modelo de Transformação foi definido.

Duas transformações de modelos foram definidas e implementadas com base no método proposto: a MT-AMP e a MT-PROAPES. Essas transformações estão voltadas para a modelagem estática (MT-AMP) e comportamental (MT-PROAPES) de um sistema de *software* embarcado baseado na plataforma: RTOS X *Real-Time Kernel* em arquiteturas ARM. A linguagem selecionada para definir essas transformações foi a ATL, uma linguagem baseada em regras e largamente reconhecida como uma solução de transformação de modelos (KALNINA *et al*., 2012; BÜTTNER *et al*., 2011; TOLOSA *et al*., 2011; TROYA; VALLECILLO, 2011; GOGOLLA e VALLECILLO, 2011). A ATL é uma linguagem de transformação de modelos relativamente nova e que apresenta importantes desafios e limitações a serem explorados (VARA *et al*., 2009; CORTELLESSA *et al*., 2008).

A verificação das transformações de modelos é essencial devido ao importante papel que essas desempenham em MDA. O teste funcional foi utilizado para a verificação das transformações implementadas. Esse teste consiste em executar a aplicação de um conjunto de casos de teste a fim de detectar possíveis erros. A geração de casos de teste envolveu a criação dos modelos de teste e a definição de critérios de teste.

Na busca por respostas para algumas questões desta pesquisa, um *survey* foi realizado para definir o estado atual da prática da engenharia de *software* na indústria de desenvolvimento de *software* embarcado. A análise dos resultados obtidos pelo *survey* possibilitou avaliar o impacto da aplicação da abordagem MDA em termos de benefícios obtidos no desenvolvimento de *software* embarcado no Brasil.

Finalmente, para a verificação e testes das transformações implementadas, casos de estudo foram realizados. A plataforma selecionada para a realização dos casos de estudo foi o RTOS *X Real-Time Kernel* em processadores ARM. O foco sobre problemas reais permitiu à autora testar a aplicação do método proposto.

6.4 DISSEMINAÇÃO DOS RESULTADOS

Os resultados desta pesquisa foram disseminados por meio de diversos artigos escritos e submetidos para periódicos e eventos da área. Até o presente momento, alguns desses artigos foram publicados, outros aceitos para publicação e alguns se encontram em processo de revisão. Cada artigo aborda uma parte importante desta pesquisa e será citado a seguir.

O método PI-MT, proposto nesta tese, foi disseminado no artigo "*A Method for the Creation of Generic Model Transformations*", apresentado e publicado nos anais da *21st International Conference on Software Engineering and Data Engineering* (SEDE 2012), em Los Angeles, Califórnia, EUA, em junho de 2012 (AGNERa *et al*., 2012). Os resultados de um comparativo realizado sobre as técnicas de refinamento em ATL foram apesentados no artigo "*Model refinement in the model driven architecture contexto*", publicado no *Journal of Computer Science*, em agosto de 2012 (AGNERb *et al*., 2012).

A transformação MT-PROAPES, definida com base no método PI-MT e que utiliza o perfil PROAPES, foi proposta no artigo "*Modeling of Embedded Software on MDA Platform Models"*, publicado no *Journal of Computer Science and Technology* (JCS&T), em outubro de 2012 (SOARESb *et al*., 2012). Este artigo apresenta a transformação MT-PROAPES, desenvolvida nesta tese, juntamente com a definição do perfil PROAPES, proposto pela pesquisadora Inali Wisniewski Soares.

A transformação de modelos MT-AMP, definida com base no método PI-MT, foi apresentada no artigo "*A Generic Model Transformation for Embedded Software based on Real-Time Operating Systems*", submetido ao *International Journal of Computer Systems Science & Engineering* (IJCSSE), em janeiro de 2012. Este artigo encontra-se em processo de revisão.

Uma revisão da literatura sobre os principais conceitos envolvidos com transformações de modelos foi apresentada no resumo expandido "*Transformação de Modelos no Contexto da MDA*", publicado nos anais da II SIEPE – Semana de Integração Ensino, Pesquisa e Extensão, realizada na Universidade Estadual do Centro-Oeste (UNICENTRO) em setembro de 2011 (AGNER *et al*., 2011).

Os resultados obtidos com a realização do survey foram disseminados por meio do artigo "*A Brazilian Survey on UML and Model-Driven Practices for Embedded Software Development*", aceito para publicação no *Journal of Systems and Software* (AGNERc *et al*., 2012)

Outro artigo relacionado com esta pesquisa, intitulado *"A Method for the development of Platform Models in the Model Driven Architecture context"*, foi publicado no *Journal of Computer Science* em novembro de 2012 (SOARESa *et al*., 2012). Este artigo apresenta o método proposto pela pesquisadora Inali Wisniewski Soares para a criação de modelos de plataforma explicitamente definidos. Ainda, o resumo expandido intitulado "*Definição de Modelos de Plataforma para o Desenvolvimento de Software Embarcado no Contexto da MDA*" foi publicado nos anais da II SIEPE – Semana de Integração Ensino, Pesquisa e Extensão, realizada na Universidade Estadual do Centro-Oeste (UNICENTRO) em setembro de 2011 (SOARES *et al*., 2011). Este resumo trata da definição e uso de Modelos de Plataforma explicitamente definidos, no âmbito da MDA.

Finalmente, o artigo "*Application of Platform Models in Model Driven Engineering of Embedded Software*" trata da proposição de um perfil de plataforma e, relacionado especificamente com esta pesquisa aborda a questão da transformação de modelos neste contexto. Esse artigo foi submetido ao *IEICE Transactions* em julho de 2012 e encontra-se em avaliação.

6.5 TRABALHOS FUTUROS

Explorar a perspectiva de implementação do método PI-MT em termos de diferentes RTOSs é considerado um tema para trabalhos futuros. Para tal, um novo perfil (metamodelo) de plataforma deve ser utilizado como base na definição das transformações implementadas. Vale ressaltar que, nesse caso o método não muda, apenas as transformações mudam se um novo perfil de plataforma for utilizado, ou seja, o método PI-MT é independente do RTOS selecionado.

À medida que a complexidade e o tamanho das transformações de modelo crescem, aumenta a demanda por mecanismos e ferramentas para testar a sua corretude. Em uma transformação de modelos é necessário, além de depurar as regras definidas, verificar se os modelos produzidos estão em conformidade com o metamodelo alvo e se as propriedades essenciais são preservadas pela transformação.

Um dos grandes desafios para realizar os testes de transformações de modelo é a heterogeneidade das linguagens de transformação de modelos e técnicas existentes (BAUDRY *et al*., 2010). Esse problema é agravado quando as transformações do modelo são definidas como uma composição de transformações de modelo encadeadas, como é o caso

desta tese. Desse modo, um dos trabalhos futuros propostos é estudar novas técnicas de verificação e testes de transformações de modelos a serem utilizadas para testar as transformações de modelos MT-AMP e MT-PROAPES realizadas nesta pesquisa. Dentre as possíveis técnicas destaca-se a USE (*UML-based Specification Environment*), que propõe o uso de "*contratos*" para verificar as transformações de modelo. Os "*contratos*" expressos em OCL visam especificar as propriedades que necessitam ser verificadas para determinada transformação (CARIOU *et al*., 2011; GOGOLLA e VALLECILLO, 2011).

As transformações MT-AMP e MT-PROAPES foram implementadas por meio da ATL. Embora a ATL seja a linguagem mais difundida e usada atualmente, foram encontradas dificuldades e limitações relacionadas com esta linguagem. Desse modo, um trabalho futuro consiste em estudar outras linguagens de transformação de modelos e implementar as referidas transformações nessa nova linguagem. Por fim, será possível realizar um comparativo entre a ATL e a nova linguagem utilizada, ressaltando os benefícios e as desvantagens de cada uma. Na literatura destacam-se as seguintes linguagens de transformação baseadas em regras e que podem ser consideradas nesse sentido: Epsilon (KOLOVOS *et al*., 2008) e MT (TRATT, 2006).

Outra sugestão de trabalho futuro é a criação de restrições OCL, por meio de regras OCL, com o objetivo de validar o modelo PIM com relação ao preenchimento de propriedades relacionadas com os estereótipos de ligação, especialmente aqueles que contêm informações fundamentais sobre a plataforma adotada. Desse modo, é possível garantir que o PIM seja definido em conformidade com os requisitos exigidos pela transformação. Restrições definidas nos modelos de entrada são chamadas de "precondições da transformação de modelos" (SEN *et al*., 2008). Uma precondição define o conjunto de modelos válidos para a transformação e é expressa por meio da linguagem OCL. Desse modo, as precondições devem especificar os requisitos a serem cumpridos pelos modelos de entrada e podem ser utilizadas para a construção de casos de teste voltados para detectar erros ocorridos devido à violação dessas restrições.

Outro tema para trabalho futuro é o uso de ontologias para descrever o método de transformação PI-MT. O uso de ontologias fornece um vocabulário comum do domínio, bem como possibilita expressar explicitamente as regras de domínio úteis em um projeto, restringindo a sua interpretação através da utilização de axiomas adequados (GUARINO, 1998). Atualmente, a infraestrutura base da *MDA* fornece uma arquitetura que possibilita: i) a criação de modelos e metamodelos; ii) a definição de transformações entre esses modelos; e iii) o gerenciamento de metadados. Embora a semântica de um modelo seja estruturalmente

bem definida por seu metamodelo, os mecanismos para descrever a semântica do domínio são bastante limitados em comparação com linguagens de representação de conhecimento. Desse modo, a definição de métodos baseados em MDA pode se beneficiar da integração com linguagens de ontologia de diversas formas, por exemplo, reduzindo a ambiguidade nas definições dos conceitos propostos (HAPPEL e SEEDORF, 2006) e facilitando a integração de informações e a interoperabilidade entre contextos (GUARINO, 1998).

REFERÊNCIAS

AGNERa, L.T.W.; SOARES, I.W.; STADSIZS, P.C., SIMAO, J.M. **PI-MT: A Method for the Creation of Generic Model Transformations**. 21st International Conference on *Software* Engineering and Data Engineering (SEDE 2012), em Los Angeles, Califórnia, EUA, p. 61-66, 2012.

AGNERb, L.T.W.; SOARES, I.W.; STADZISZ, P.C.; SIMAO, J.M. **Model refinement in the model driven architecture context**. Journal of Computer Science, v. 8, n. 8, p. 1205-1211, 2012. DOI: 10.3844/jcssp.2012.1205.1211.

AGNERc, L.T.W.; SOARES, I.W.; STADZISZ, P.C.; SIMAO, J.M. **A Brazilian Survey on UML and Model-Driven Practices for Embedded Software Development**. Journal of Systems and *Software*, aceito para publicação, 2012. DOI: 10.1016/j.jss.2012.11.023.

AGNER, L.T.W.; SOARES, I.W.; STADZISZ, P.C.; SIMAO, J.M. **Transformação de Modelos no Contexto da MDA**. Anais da II SIEPE – Semana de Integração Ensino, Pesquisa e Extensão, UNICENTRO, 27 a 29 de setembro, 2011.

AGNER, L.T.W. **Proposta de um Método para Transformação de Modelos no Contexto da MDA**. Qualificação de Doutorado, Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, 2010.

ALLILAIRE, F.; BEZIVIN, J.; JOUAULT, F.; KURTEV, I. **ATL: Eclipse support for model transformation**. Eclipse Technology eXchange Workshop (eTX) at the European Conference on Object-Oriented Programming (ECOOP' 06), Nantes, France, 2006.

ALMEIDA, J.P.A. **Model-Driven Design of Distributed Applications**. Tese de Doutorado em Ciência da Computação, CTIT Ph.D. Thesis Series, No. 06-85, Telematica Instituut Fundamental Research Series, No. 018 (TI/FRS/018). 2006.

ALTI, A.; KHAMMACI, T.; SMEDA, A. **Integrating Software Architecture Concepts into the MDA Platform with UML Profile**. Journal of Computer Science, v. 3, p. 793-802, 2007.

AMATO NETO, J. **As políticas industriais e tecnológicas e as pequenas e médias empresas: iniciativas na história recente da economia brasileira**. Revista Gestão Industrial, v. 3, n. 1, p. 87-102, 2007.

AMELLER, D.; FRANCH, X.; CABOT, J. **Dealing with Non-Functional Requirements in Model-Driven Development**. 18th IEEE International Requirements Engineering Conference (RE' 10), Sydney, NSW , p. 189 – 198, 2010.

ANASTASAKIS, K.; BORDBAR, B.; GEORG, G.; RAY, I. **On challenges of model transformation from UML to Alloy**. Software and Systems Modeling, v. 9, n. 1, p. 69-86, 2010.

AROCA, R.V.; CAURIN, G. **A Real Time Operating Systems (RTOS) Comparison**. VI Workshop de Sistemas Operacionais (WSO'2009): Evento satélite do XXIX Congresso da Sociedade Brasileira de Computação (SBC´2009), Bento Gonçalves, RS, 2009.

ATKINSON, C.; KÜHNE, T. **Model-Driven Development: A Metamodeling Foundation**. IEEE *Software*, v. 20, n. 5, p. 36-41, 2003.

ATMEL. **Atmel Corporation**. Disponível em: <http://www.atmel.com/>. Acesso em: Março de 2010.

BALASUBRAMANIAN, K.; GOKHALE, A.; KARSAI, G.; SZTIPANOVITS, J.; NEEMA, S. **Developing applications using model-driven design environments**. IEEE Computer, v. 39, n. 2, p. 33- 40, 2006.

BASHA, N.M.J., MOIZ, S.A., RIZWANULLAH, M. **Model Based Software Development: Issues & Challenges**. International Journal of Computer Science & Informatics (IJCSI), v. 2, n. 1-2, p. 2231–5292, 2012.

BAUDRY, B.; GHOSH, S.; FLEUREY, F.; FRANCE, R.; TRAON, Y.L.; MOTTU, J.-M. **Barriers to systematic model transformation testing**. Communications of the ACM, v. 53, p. 139-143, ACM Press, 2010.

BECKER, M.; DI GUGLIELMO, G.; FUMMI, F.; MUELLER, W.; PRAVADELLI, G.; XIE, T. **RTOS-aware refinement for TLM2.0-based HW/SW designs**. Conference on Design, Automation and Test in Europe (DATE' 10), European Design and Automation Association, Bélgica, p. 1053-1058, 2010.

BEHNAM, M.; NOLTE, T.; SHIN, I.; ASBERG, M.; BRIL, R.J. **Towards hierarchical scheduling on top of VxWorks**. 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 08), p. 63-72, 2008.

BERGER, A. **Embedded Systems Design: An Introduction to processes, tools and techniques**. CMP Books, 2002.

BERTHOMIEU, B., BODEVEIX, J.P., FARAIL, P., FILALI, M., GARAVEL, H., GAUFILLET, P., LANG, F., VERNADAT, F. **FIACRE: an intermediate language for model verification in the TopCased environment**. 4th European Congress on Embedded Real-Time *Software* (ERTS'08), Toulose, Janeiro de 2008.

BÉZIVIN, J.; BÜTTNER, F.; GOGOLLA, M.; JOUAULT, F.; KURTEV, I.; LINDOW, A. **Model Transformations? Transformation Models!** 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS' 06), 2006.

BRIAND, L. C.; LABICHE, Y.; YUE, T. **Automated traceability analysis for UML model refinements**. Information and *Software* Technology, v. 51, n. 2, p. 512-527, 2009.

BUTENHOF, D.R. **Programming with POSIX Threads**. Addison-Wesley, 1997.

BÜTTNER, J. F.; CABOT, J.; GOGOLLA, M. **On validation of ATL transformation rules by transformation models**. 8th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa' 11). ACM, New York, NY, USA, Article 9 , p. 1-8, 2011.

CARIOU, E.; BALLAGNY, C.; FEUGAS, A.; BARBIER, F. **Contracts for model execution verification**. European Conference on. Modelling Foundations and Applications (ECMFA 2011). LNCS, Springer, v. 6698, p. 3-18, 2011.

CHEHADE, W.E.H.; RADERMACHER, A.; TERRIER, F.; SELIC, B.; GÉRARD, S. **A Model-Driven Framework for the Development of Portable Real-Time Embedded Systems**. 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS' 2011), p. 45-54, 2011.

CICCHETTI, A.; DI RUSCIO, D.; PIERANTONIO, A. **A Metamodel Independent Approach to Difference Representation**. Journal of Object Technology, v. 6, n. 9, p. 165-185, 2007.

CORTELLESSA, V.; DI GREGORIO, S.; DI MARCO, A. **Using ATL for transformations in software performance engineering: a step ahead of java-based transformations?**. 7th International Workshop on Software and Performance (WOSP '08), New York, NY, USA, ACM, p. 127-132, 2008.

CRÉGUT, X., COMBEMALE, B., PANTEL, M., FAUDOUX, R., PAVEI, J. **Generative technologies for model animation in the TopCased platform**. 6th European Conference on Modelling Foundations and Applications (ECMFA' 10), Paris, França. LNCS, Springer, v. 6138, p. 90-103, 2010.

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

DANG, D.-H.; GOGOLLA, M. **Precise Model-Driven Transformations Based on Graphs and Metamodels**. 7th IEEE International Conference on *Software* Engineering and Formal Methods (SEFM' 09), IEEE Xplore, p. 307-316, 2009.

DEKEYSER, J.; BOULET, P.; MARQUET, P.; MEFTALI, S. **Model driven engineering for SoC co-design**. The 3rd International IEEE-NEWCAS Conference, p. 21- 25, 2005.

DOBING, B.; PARSONS, J. **Dimensions of UML Diagram Use: A Survey of Practitioners**. Journal of Database Management (JDM'08), v. 19, n. 1, p. 1-18, 2008.

DOUGLASS, B.P. **Doing Hard Time: Developing real-time systems with UML, objects, frameworks, and patterns**. The Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

DUBE, M.R.; DIXIT, S.K. **Modeling Theories and Model Transformation Scenario for Complex System Development**. International Journal of Computer Applications, v. 38, n. 7, p. 11-18, 2012.

EBERT, C.; JONES, C. **Embedded Software: Facts, Figures, and Future**. IEEE Computer, v. 42, n. 4, p. 42-52, 2009.

ECLIPSE. **ATL/User Guide - The ATL Language. The Eclipse Foundation**. Acesso em: março de 2012. http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language.

ECLIPSE. **ATL/User Guide - Overview of the Atlas Transformation Language**. The Eclipse Foundation. Acesso em: março de 2012. http://wiki.eclipse.org/ATL/User_Guide_-_Overview_of_the_Atlas_Transformation_Language.

EHRIG, H.; HOFFMANN, K.; PADBERG, J. **Transformations of Petri Nets**. Electronic Notes in Theoretical Computer Science, v. 148, n. 1, p. 151-172, 2006.

ERIKSSON, H.; PENKER, M.; LYONS, B.; FADO, D. **UML 2 Toolkit**. Wiley Publishing, 2004. 552p.

ESPINOZA, H.; SELIC, B.; CANCILA, D.; GÉRARD, S. **Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems**. European Conference on Model Driven-Architecture Foundations and Applications (ECMDA' 09). LNCS, v. 5562, p. 98-113, 2009.

ESYSTECH. **X Real-Time Kernel**. eSysTech Embedded System Technologies. Disponível em: <http://www.esystech.com.br/produtos/XKernel/XKernel.php>. Acesso em: Julho de 2012.

FARAIL, P. **Toolkit in OPen-source for Critical Applications & SystEms Development**. Aerospace Valley Presentation, www.topcased.org.

FERNÁNDEZ-FERNÁNDEZ, H.; PALACIOS-GONZÁLEZ, E.; GARCÍA-DÍAZ, V.; GARCIA-BUSTELO, B.C.P.; LOVELLE, J.M.C. **Design of Intelligent Business Applications based in BPM and MDE**. International Conference on Artificial Intelligence (IC-AI' 2008), p. 591–597, Las Vegas, Nevada, USA, CSREA Press, 2008.

FINK, A. **How to sample in surveys**. Thousand Oaks, Sage, 1995.

FIORENTINI, C., MOMIGLIANO, A., ORNAGHI, M., POERNOMO, I. **A constructive approach to testing model transformations**. 3rd International Conference Theory and Practice of Model Transformations (ICMT 2010). LNCS, Springer, v. 6142, p. 77-92, 2010.

FLEUREY, F., BAUDRY, B., MULLER, P.A., TRAON, Y.L. **Qualifying input test data for model transformations**. Software and System Modeling, v. 8, n. 2, p. 185-203, 2009.

FORWARD, A.; LETHBRIDGE, T.C. **Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals**. International Workshop on Models in *Software* Engineering (MiSE' 08), New York, NY, USA. ACM Press, p. 27-32, 2008.

FOWLER, F.J. **Improving survey questions: Design and evaluation**. Thousand Oaks, CA: Sage Publications, 1995.

FRANCE, R.; RUMPE, B. **Model-driven Development of Complex Software: A Research Roadmap**. Future of *Software* Engineering (FOSE '07), Washington, DC, USA, IEEE Computer Society, p. 37-54, 2007.

FUENTES-FERNÁNDEZ, L.; VALLECILLO-MORENO, A. **An introduction to UML profiles**. European Journal for the Informatics Professional, v. 5, n. 2, p. 5–13, 2004.

GALVÃO, I.; GOKNIL, A. **Survey of Traceability Approaches in Model-Driven Engineering**. 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), p. 313-324, 2007.

GERARD, S., PETRIU, D., MEDINA, J. **MARTE: A New Standard for Modeling and Analysis of Real-Time and Embedded Systems**. 19th Euromicro Conference on Real-Time Systems (ECRTS 07), Pisa, Itália, 2007.

GERARD, S., BABAU, J.P., CHAMPEAU, J. **Model Driven Engineering for Distributed Real-Time Embedded Systems**. Wiley-IEEE Press, 2010.

GHERBI, T.; MESLATI, D.; BORNE, I. **MDE between Promises and Challenges**. 11th International Conference on Computer Modelling and Simulation (UKSIM '09), p. 152-155, 2009.

GÓES, R.; RENAUX, D.P.B. **Desenvolvimento de um Sistema Operacional Orientado a Objetos para uso em Sistemas Embarcados**. III Workshop de Sistemas Operacionais, 2006, Campo Grande - MS. Anais do XXVI Congresso da SBC, p. 21-30, 2006.

GOGOLLA, M.; VALLECILLO, A. **Tractable Model Transformation Testing**. Modelling Foundations and Applications, LNCS, Springer, v. 6698, p. 221-235, 2011.

GOLDSCHMIDT, T.; WACHSMUTH, G. **Refinement transformation support for QVT Relational transformations**. 3rd Workshop on Model Driven *Software* Engineering (MDSE' 08), Berlin, Germany, 2008.

GRAY, J.; ZHANG, J.; LIN, Y.; ROYCHOUDHURY, S.; WU, H.; SUDARSAN, R.; GOKHALE, A.; NEEMA, S.; SHI, F.; BAPTY, T. **Model-Driven Program Transformation of a Large Avionics Framework**. Generative Programming and Component Engineering (GPCE 2004), LNCS, Springer, v. 3286, p. 361-378, 2004.

GRONMO, R.; MOLLER-PEDERSEN, B.; OLSEN, G.K. **Comparison of Three Model Transformation Languages**. 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA' 09), LNCS, Springer, v. 5562, p. 2-17, 2009.

GROSSMAN, M.; ARONSON, J.E.; MCCARTHY, R.V. **Does UML make the grade? Insights from the software development community**. Information and *Software* Technology, v. 47, n. 6, p. 383-397, 2005.

GUARINO, N. **Formal ontology and information systems**. 1st International Conference on Formal Ontology in Information Systems (FOIS 1998), Trento, Itália. Amsterdam: IOS Press, p. 3-15, 1998.

GUEDES, G.T.A. **UML2: Uma abordagem prática**. Novatec Editora, São Paulo, 2009.

GUERRA, E. **Specification-driven test generation for model transformations**. 5th International Conference Theory and Practice of Model Transformations (ICMT 2012), LNCS, Springer, v. 7307, p. 40-55, 2012.

GUERRA, E.; LARA, J.; KOLOVOS, D.; PAIGE, R.; SANTOS, O. **Engineering model transformations with transML**. *Software* and Systems Modeling, p. 1-23, 2011.

HAPPEL, H.J., SEEDORF, S. **Applications of ontologies in software engineering**. 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), 2006, Athens, EUA.

HERRINGTON, J. **Code Generation in Action**. Manning Publications Co., Greenwich, 2003.

HWANG, J.-Y.; SUH, S.-B.; HEO, S.-K.; PARK, C.-J.; RYU, J.-M.; PARK, S.-Y.; KIM, C.-R. **Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones**. 5th IEEE Consumer Communications and Networking Conference (CCNC' 08). IEEE Computer Society, p. 257-261, 2008.

HUTCHINSON, J.; WHITTLE, J.; ROUNCEFIELD, M.; KRISTOFFERSEN, S. **Empirical assessment of MDE in industry**. 33rd International Conference on *Software* Engineering (ICSE' 11). ACM Press, p. 471-480, 2011.

IBGE. **Tabela 2.1 - Variáveis selecionadas das empresas das indústrias extrativas e de transformação, segundo as Grandes Regiões e Unidades da Federação selecionadas – 2006 - 2008**. Levantamento de Inovação Tecnológica, Instituto Brasileiro de Geografia e Estatítistica (IBGE), 2008.

JELITSHKA, A.; CIOLKOWSKI, M.; DENGER, C.; FREIMUT, B.; SCHLICHTING, A. **Relevant information sources for successful technology transfer: a survey using inspections as an example**. First International Symposium on Empirical *Software* Engineering and Measurement (ESEM' 07). IEEE Computer Society, p. 31-40, 2007.

JEON, S.U.; HONG, J.E.; SONG, I.G.; BAE, D.H. **Developing platform specific model for MPSoC architecture from UML-based embedded software models**. Journal of Systems and *Software*, v. 82, n. 10, p. 1695-1708, 2009.

JIE, H.; GEN-BAO, Z. **High-performance embedded processor technology**. International Conference on Computer Design and Applications (ICCDA' 10), vol. 1, p. V1-87-V1-89, 2010.

JOUAULT, F.; KURTEV, I. **Transforming models with ATL**. Model Transformations in Practice Workshop at MoDELS' 2005, Montego Bay, Jamaica, 2005. LNCS, Springer, v. 3844, p. 128-138, 2006.

JOUAULT, F.; ALLILAIRE, F.; BÉZIVIN, J.; KURTEV, I. **ATL: A model transformation tool**. Science of Computer Programming, v. 72, n. 1-2, p. 31–39, 2008.

KALNINA, E.; KALNINS, A.; SOSTAKS, A.; CELMS, E.; IRAIDS, J. **Tree Based Domain-Specific Mapping Languages**. SOFSEM 2012: Theory and Practice of Computer Science. LNCS, Springer Berlin, v. 7147, p. 492-504, 2012.

KALNINS, A.; BARZDINS, J.; CELMS, E. **Model transformation language MOLA**. Model Driven Architecture: Foundations and Applications (MDAFA' 04), Linkoping, Sweden, p. 14-28, 2004.

KARSAI, G.; NEEMA, S.; SHARP, D. **Model-driven architecture for embedded software: A synopsis and an example**. Science of Computer Programming, v. 73, n. 1, p. 26-38, 2008.

KLEPPE, A.; WARMER, J.; BAST, W. **MDA Explained: The Model Driven Architecture Practice and Promise**. Addison Wesley, 2003.

KOLOVOS, D.S.; PAIGE, R.F.; POLACK, F. **The Epsilon transformation language**. International Conference on Model Transformation, Zurich, Switzerland. LNCS, Springer, v. 5063, p. 46-60, 2008.

KOPETZ, H. **The Complexity Challenge in Embedded System Design**. 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC' 08), p. 3-12, 2008.

KRTEN, R. **Getting Started with QNX Neutrino: A Guide for Realtime Programmers**. QNX *Software* Systems GmbH & Co. KG., 2009.

KUKKALA, P.; RIIHIMÂKI, J.; HAMALAINEN, M.; KRONLOF, K. **UML 2.0 Profile for Embedded System Design**. Automation and Test in Europe Conference (DATE' 05), p. 710-715, 2005.

KURTEV, I. **Adaptability of Model Transformations**. Tese de Doutorado, Universidade de Twente, 2005. ISBN 90-365-2184-X.

KÜSTER, J.M. **Systematic Validation of Model Transformations**. 3rd UML Workshop in Software Model Engineering (WiSME' 2004), 2004.

KÜSTER, J.M.; ABD-EL-RAZIK, M. **Validation of model transformations - first experiences using a white box approach**. Model Design and Validation Workshop associated to MoDELS'06, Genova, Itália, 2006. LNCS, Springer-Heidelberg, v. 4364, p. 193-204, 2006.

LAFI, L.; HAMMOUDI, S.; FEKI, J. **Metamodel Matching Techniques in MDA: Challenge, Issues and Comparison**. Model and Data Engineering, LNCS, Springer, v. 6918, p. 278-286, 2011.

LAMARI, M. **Towards an automated test generation for the verification of model transformations**. ACM Symposium on Applied Computing, p. 998-1005, Seoul, Korea, 2007.

LEAL, G.C. **Uma Abordagem Integrada de Desevolvimento e Teste de Software para Equipes Distribuídas**. Dissertação, Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, Maringá, 2010.

LEE, E.A. **Embedded Soft**. Advances in Computers, vol. 56, Academic Press, 2002.

LOPES, D.; HAMMOUDI, S.; BÉZIVIN, J.; JOUAULT, F. **Mapping Specification in MDA: From Theory to Practice**. First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA' 05), Geneva, Suíça, 2005.

LOUHICHI, S.; GRAIET, M.; KMIMECH, M.; BHIRI, M.T.; GAALOUL, W.; CARIOU, E. **ATL Transformation of UML 2.0 for the Generation of SCA Model**. International Conference on *Software* Engineering Advances (ICSEA' 11), Barcelona, Spain, 2011.

MAENG, J.; KIM, J.H.; RYU, M. **An RTOS API translator for model-driven embedded software development**. 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA' 06), Sydney, Australia, p. 363-367, 2006.

MAFRA, S.N.; BARCELOS, R.F.H. **Aplicando uma metodologia baseada em evidências na e denifição de novas tecnologias de software**. XX Simpósio Brasileiro de Engenharia de Software (SBES), Florianopólis, SC, 2006, p. 239-254.

MARWEDEL, P. **Embedded System Design**. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

MCQUILLAN, J.A.; POWER, J.F. **White-Box Coverage Criteria for Model Transformations**. 1st International Workshop on Model Transformation with ATL (MtATL 2009), Nantes, França, p. 63-77, 2009.

MELLOR, S.J.; SCOTT, K.; UHL, A.; WEISE, D. **MDA Destilada: Princípios de Arquitetura Orientada por Modelos**. Editora Ciência Moderna, 2005.

MENS, T.; VAN GORP, P. **A Taxonomy of Model Transformation**. Electronic Notes in Theoretical Computer Science, v. 152, p. 125-142, 2006.

MENTOR GRAPHICS. **Nucleus: Datasheet**. Mentor Graphics. Disponível em: <http://www.mentor.com/embedded-*software*/upload/Nucleus_OS_Brochure.pdf>. Acesso em: Julho de 2012.

MOHAGHEGHI, P.; DEHLEN, V. **Where is the Proof? A Review of Experiences from Applying MDE in Industry**. In: European Conference on Model Driven-Architecture Foundations and Applications (ECMDA-FA) 2008. LNCS, Springer, v. 5095, p. 432-443, 2008.

NOERGAARD, T. **Embedded System Architecture**. Elsevier Inc., 2005.

NUGROHO, A.; CHAUDRON, M.R.V. **A survey into the rigor of UML use and its perceived impact on quality and productivity**. 2nd International Symposium of Empirical *Software* Engineering and Measurement (ESEM' 08), IEEE Computer Society, p. 90-99, 2008.

OMG. **Model Driven Architeture (MDA)**. Object Management Group, 2001. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

OMG. **MDA Guide Version 1.0.1**. Object Management Group, 2003. www.omg.org/cgi-bin/doc?omg/03-06-01.

OMG. **Object Constraint Language: Version 2.2**. Object Management Group, 2010. http://www.omg.org/spec/OCL/2.2/PDF/

OMG. **Unified Modeling Language (UML): Superstructure**. Object Management Group, 2011a. http://www.omg.org/spec/UML/2.4/Superstructure/Beta2/PDF.

OMG. **Meta Object Facility (MOF) Core Specification: Version 2.4.1**. Object Management Group, 2011b. http://www.omg.org/spec/MOF/2.4.1/PDF/.

OSTRAND, T.J., BALCER, M.J. **The category-partition method for specifying and generating functional tests**. Communications of the ACM, v. 31, n. 6, p. 676-686, 1988.

PERRIEN, J.; CHÉRON, E.J.; ZINS, M. **Recherche en marketing: méthodes et décisions**. Montreal: Gaëtan Morin Editeur, 1984.

PONS, C. GARCIA, D. **A Lightweight Approach for the Semantic Validation of Model Refinements**. Electronic Notes in Theoretical Computer Science, v. 220, n. 1, p. 43-61, 2008.

PONTISSO, N., CHEMOUIL, D. **TOPCASED Combining Formal Methods with Model-Driven Engineering**. 21st IEEE/ACM International Conference on Automated *Software* Engineering (ASE' 06), p. 359-360, 2006.

PRASHANTH, C.M.; SHET, K.C. **Efficient Algorithms for Verification of UML Statechart Models**. Journal of *Software*, v. 4, n.3, p. 175-182, 2009.

ROSS, D.T. **Applications and Extensions of SADT**. IEEE Computer, v. 18, n. 4, p. 25-34, 1985.

SALEM, A.K.B.; OTHMAN, S.B.; ABDELKRIM, H.; SAOUD, S.B. **RTOS for SoC embedded control applications**. 3rd International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS' 2008), p.1-6, 2008.

SANDRIESER, M.; BENKNER, S.; PLLANA, S. **Explicit Platform Descriptions for Heterogeneous Many-Core Architectures**. 16th International Workshop on High-Level

Parallel Programming Models and Supportive Environments (HIPS 2011), IPDPS Workshops 2011, Alaska, USA, IEEE Computer Society, p. 1292-1299, 2011.

SELIC, B. **The Pragmatics of Model-Driven Development**. IEEE *Software*, v. 20, n. 5, p. 19-25, 2003.

SELIC, B. **On software platforms, their modeling with UML 2, and platform-independent design**. 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC' 05), Seattle, WA, USA, ACM Press, p. 15-21, 2005.

SEN, S., BAUDRY, B., MOTTU, J.M. **On combining mullti-formalism knowledge to select test models for model transformaion testing**. ACM/IEEE International Conference on Software Testing (ICST 2008), Lillehammer, Noruega, 2008.

SEN, S., BAUDRY, B., MOTTU, J.M. **Automatic Model Generation Strategies for Model Transformation Testing**. 2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009). LNCS, Springer, v. 5563, p. 148-164, 2009.

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

SHULL, F.; CARVER, J.; TRAVASSOS, G.H. **An empirical methodology for introducing software processes**. SIGSOFT Softw. Eng. Notes, v. 26, n. 5, p. 288-296, 2001.

SIEGEL, J., OMG Staff Strategy Group. **Developing in OMG's Model-Driven Architecture**. Technical Report - White Paper, Revision 2.6, Object Management Group, 2001.

SILVA, M.V.B.; SILVEIRA NETO, R.M. **Dinâmica da concentração da atividade industrial no Brasil entre 1994 e 2004: uma análise a partir de economias de aglomeração e da nova geografia econômica**. Economia Aplicada, v. 13, n. 2, p. 299-331, 2009.

SILVESTRE, E.A. **Modelagem de Software de Tempo-Real Utilizando o Profile MARTE da UML**. Dissertação de Mestrado. Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, 2012.

SINGH, Y.; SOOD, M. **Model Driven Architecture: A perspective**. IEEE International Advance Computing Conference (IACC 2009). IEEE Xplore Press, p. 1644-1652, 2009.

SPARK SYSTEMS. **Enterprise Architect: User Guide**. 2012. Disponível em: http://www.sparxsystems.com/enterprise_architect_user_guide/9.3/index.html.

STADZISZ, P.C.; RENAUX, D.P.B. **Software Embarcado**. XIV Escola Regional de Informática (ERI 2007), Universidade Estadual do Centro-Oeste (UNICENTRO), SBC (Organização), Paraná, Brasil, v.1, p. 107-155, 2007.

SOARES, I.W. **Método para a Construção de Modelos de Plataforma no Contexto da MDA**. Tese de Doutorado, Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná, 2012.

SOARESa, I.W., AGNER, L.T.W, STADZISZ, P.C., SIMÃO, J.M. **A Method for the development of Platform Models in the Model Driven Architecture context**. Journal of Computer Science, vol. 8, p. 1932-1939, 2012.

SOARESb, I.W., AGNER, L.T.W, STADZISZ, P.C., SIMÃO, J.M. **Modeling of Embedded Software on MDA Platform Models**. Journal of Computer Science and Technology (JCS&T), vol. 12, p. 133-139, 2012.

SOARES, I.W. ; AGNER, L.T.W.; STADZISZ, P.C.; SIMÃO, J.M. **Definição de Modelos de Plataforma para o Desenvolvimento de Software Embarcado no Contexto da MDA**. Anais da II SIEPE – Semana de Integração Ensino, Pesquisa e Extensão, UNICENTRO, 27 a 29 de setembro, 2011.

SOMMERVILLE, I. **Engenharia de Software**, 6ª ed. Pearson Addison Wesley, 2003.

SUN, Y.; WHITE, J.; GRAY, J. **Model transformation by demonstration**. Model Driven Engineering Languages and Systems (MoDELS' 09). LNCS, Springer, v. 5795, p. 712-726, 2009.

TAENTZER, G.; MÜLLER, D.; MENS, T. **Specifying Domain-Specific Refactorings for AndroMDA Based on Graph Transformation**. Applications of Graph Transformations with Industrial Relevance, LNCS, Springer, v. 5088, p. 104-119, 2008.

TAO, Y.; SONG, K. **Design of VxWorks-based software architecture for space optical remote sensor**. International Conference on Electronic and Mechanical Engineering and Information Technology (EMEIT' 11), v. 2, p. 828-831, 2011.

TEPPOLA, S.; PARVIAINEN, P.; TAKALO, J. **Challenges in Deployment of Model Driven Development**. Fourth International Conference on *Software* Engineering Advances (ICSEA' 09), p. 15-20, 2009.

THOMAS, F.; GÉRARD, S.; DELATOUR, J.; TERRIER, F. **Software real-time resource modeling**. International Conference Forum on Specification and Design Languages (FDL'07), LNCS, Springer, v. 10, p. 169-182, 2007.

TISI, M.; MARTÍNEZ, S.; JOUAULT, F.; CABOT, J. **Refining Models with Rule-based Model Transformations**. Technical Report, AtlanMod, INRIA & École des Mines de Nantes, 2011.

TOLOSA, J.B.; SANJUÁN-MARTÍNEZ, O.; GARCÍA-DÍAZ, V.; G-BUSTELO, B.C.P.; LOVELLE, J.M.C. **Towards the systematic measurement of ATL transformation models**. *Software*: Practice and Experience, v. 41, p. 789–815, 2011.

TRATT, L. **The MT model transformation language**. ACM Symposium on Applied Computing (SAC' 06). ACM Press, New York, NY, USA, p. 1296-1303, 2006.

TROYA, J.; VALLECILLO, A. **A Rewriting Logic Semantics for ATL**. Journal of Object Technology, v. 10, n. 5, p. 1-29, 2011.

VAN AMSTEL, M.F; VAN DEN BRAND, M.G.J. **Using Metrics for Assessing the Quality of ATL Model Transformations**. 3rd International Workshop on Model Transformation with ATL (MtATL' 2011), Zurich, Switzerland, CEURS, v. 742, p. 20-34, 2011.

VAN AMSTEL, M.F.; BOSEMS, S.; KURTEV, I.; PIRES, L.F. **Performance in model transformations: experiments with ATL and QVT**. 4th International Conference on Theory and Practice of Model Transformations (ICMT' 11), LNCS, Springer, v. 6707, p. 198-212, 2011.

VAN DER STRAETEN, R.; JONCKERS, V.; MENS, T. **A formal approach to model refactoring and model refinement**. Software and System Modeling, v. 6, n. 2, p. 139-162, 2007.

VAN GORP, P.; MULIAWAN, O.; KELLER, A.; JANSSENS, D. **Executing a platform independent model of the UML-to-CSP transformation on a commercial platform**. AGTIVE 2007 Tool Contest, 2007.

VARA, J.M.; BOLLATI, V.A.; VELA, B.; MARCOS, E. **Leveraging model transformations by means of annotation models**. International Workshop on Model Transformation with ATL (MtATL' 09), p. 96-102, 2009.

VARRÓ, D.; VARRÓ, G.; PATARICZA, A. **Designing the Automatic Transformation of Visual Languages**. Science of Computer Programming, v. 44, n. 2, p. 205-227, 2002.

WAGELAAR, D.; JONCKERS, V. **Explicit Platform Models for MDA**. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS' 2005), Jamaica. LNCS, Springer-Verlag, v. 3713, p. 367-381, 2005.

WAGELAAR, D. **Composition techniques for rule-based model transformation languages**. International Conference on Model Transformation (ICMT'08), Zurich, Switzerland, LNCS, Springer, v. 5063, p. 152-167, 2008.

WAGELAAR, D.; VAN DER STRAETEN, R.; DERIDDER, D. **Module superimposition: a composition technique for rule-based model transformation languages**. *Software* and Systems Modeling, v. 9, n. 3, p. 285-309, 2010.

WANG, J., KIM, S.-K., CARRINGTON, D. **Automatic Generation of Test Models for Model Transformations**. 19th Australian Conference on Software Engineering (ASWEC' 2008), IEEE Xplore, p. 432-440, 2008.

WIRTH, N. **Program development by stepwise refinement**. Communications of the ACM, v. 14, n. 4, p. 221-227, 1971.

WOLF, W.H. **Computers as Components: Principles of Embedded Computer System Desgin**, 3ª ed. The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier, 2012.

YIE, A.; WAGELAAR, D. **Advanced Traceability for ATL**. 1st International Workshop on Model Transformation with ATL (MtATL'09), Nantes, France, p. 78-87, 2009.