Universidade Estadual de Londrina

WILSON HISSAMU SHIRADO

DESENVOLVIMENTO DE SISTEMAS DE AUTOMAÇÃO DE TESTE FUNCIONAIS PARA SISTEMAS EMBARCADOS DIRIGIDOS POR MODELO

Londrina
2016

WILSON HISSAMU SHIRADO

DESENVOLVIMENTO DE SISTEMAS DE AUTOMAÇÃO DE TESTE FUNCIONAIS PARA SISTEMAS EMBARCADOS DIRIGIDOS POR MODELO

Dissertação apresentado ao Curso de Mestrado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Orientadora: Profª. Drª. Jandira Guenka Palma

Londrina
2016

Shirado, Wilson Hissamu.

Desenvolvimento de Sistemas de Automação de Testes Funcionais para Sistemas Embarcados Dirigidos por Modelo / Wilson Hissamu Shirado. - Londrina, 2016.
136 f. : il.

Orientador: Jandira Guenka Palma.

Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2016.

Inclui bibliografia.

1. Sistemas Embarcados - Teses. 2. Teste de Software - Teses. 3. Automação de Testes - Teses. 4. Model Driven Architecture - Teses. I. Palma, Jandira Guenka. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

WILSON HISSAMU SHIRADO

DESENVOLVIMENTO DE SISTEMAS DE AUTOMAÇÃO DE TESTE FUNCIONAIS PARA SISTEMAS EMBARCADOS DIRIGIDOS POR MODELO

Dissertação apresentado ao Curso de Mestrado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

____

Orientadora: Profª. Drª. Jandira Guenka Palma
Universidade Estadual de Londrina - UEL

____

Prof. Dr. Evandro Baccarin
Universidade Estadual de Londrina - UEL

____

Prof. Dr. Alexandre L'Erario
Universidade Tecnológica Federal do Paraná - UTFPR

Londrina, 02 de Fevereiro de 2016.

Aos meus pais...

AGRADECIMENTOS

Primeiramente honro meus agradecimentos a esta instituição de ensino que me oportunizou a janela que hoje vislumbro de um novo horizonte. Em segundo, agradeço ao Programa de Pós-Graduação Stricto Sensu – Mestrado em Ciência da Computação que possibilitou a realização do sonho de tornar-me Mestre em Ciência da Computação. Em terceiro, mas tão relevante quanto, agradeço ao CNPq que permitiu, consentiu e disponibilizou a bolsa de estudo para o Mestrado.

Dedico um agradecimento especial a minha orientadora, Profª Dra Jandira Guenka Palma que através de seu conhecimento e dedicação, me norteou durante o árduo caminho percorrido. Obrigado pela paciência, disposição, compartilhamento e acima de tudo pela confiança depositada.

Agradeço aos demais docentes do Programa de Pós-Graduação pelo apoio e pela sabedoria concedida. E como também as amizades construídas e que se perpetuarão por anos vindouros.

Agradeço a Deus por minha vida e pela inspiração durante esta longa caminhada na busca do conhecimento.

Aos meus queridos pais que de forma especial e carinhosa deram-me força e coragem nos momentos de maior aflição e dificuldade, e a quem eu devoto todas as noites de minha existência.

Aos meus irmãos Takeshi e Yuri, que mesmo distantes sempre se fizeram presente em minha vida, concedendo a mim votos de confiança, credibilidade e amor.

Minha tia Valdira Emiko Shirado a quem considero uma segunda mãe pelo zelo e cuidado para com minha pessoa. Agradeço ainda a minha noiva Cássia Cotrim e a minha querida filha Ana Clara com quem compartilhei os momentos bons e também os momentos difíceis nesta jornada.

A todos os demais familiares que através de palavras de incentivo e gestos de amizade, apoiaram-me durante toda a jornada, não permitindo que eu perdesse a perseverança em mim mesmo.

E por fim, mas não menos admiráveis sou grato a todos aqueles que colaboraram direta ou indiretamente na conclusão deste trabalho.

“Há conhecimento de dois tipos: Sabemos sobre um assunto, ou sabemos onde podemos buscar informações sobre ele. (Samuel Johnson)”

SHIRADO, Wilson Hissamu. **Desenvolvimento de sistemas de automação de teste funcionais para sistemas embarcados dirigidos por modelo**. 2016. 136f. Dissertação de Mestrado (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2016.

RESUMO

Os sistemas embarcados estão presentes nos mais variados ramos da atividade humana, chegando ao ponto de até 98% dos processadores produzidos no mundo são alocados em aplicações deste tipo. Assim, é comum que os mesmos estejam presentes em equipamentos ou sistemas críticos, sendo portanto, a etapa de testes durante o processo de desenvolvimento de crucial importância. No entanto, ainda hoje as atividades de teste de software são pouco difundidas e empregadas de forma sistemática dentro de uma grande parcela das empresas de desenvolvimento, em parte por ser uma etapa demorada, complexa e custosa. A esses fatores, somam-se ainda a escassez de documentação de testes, problemas oriundos da variável humana no processo de testes e ainda a necessidade de realizar testes de regressão após modificações, concertos ou atualizações do produto. Frente a essa situação, muitos esforços têm sido direcionados à automatização de testes. No entanto o desenvolvimento de ferramentas para automação podem demandar elevados esforços de forma que em muitos casos as empresas optam pela realização manual dos testes. Diante deste cenário, esta proposta busca aplicar novos paradigmas de desenvolvimento, denominados desenvolvimento dirigido por modelos, conhecido como MDD (*Model Driven Development*). O MDD é um conceito que foi trabalhado pela OMG (*Object Management Group*) e a partir do qual foi proposto o MDA (*Model Driven Architecture)* na forma de uma abordagem que busca elevar o nível de abstração no desenvolvimento de softwares. Desta forma, este trabalho propôs um processo de desenvolvimento de sistemas de automação de testes funcionais para sistemas embarcados pautados na aplicação dos conceitos do MDA. Foram ainda realizados um teste experimental do processo e as análises dos resultados obtidos em relação a outras abordagens e soluções para a problemática.

Palavras-chave: Sistemas Embarcados,Teste de Software. Automação de Testes. MDA.

SHIRADO, Wilson Hissamu. **Model driven development of functional test automation for embedded systems**. 2016. 136p. Master's Thesis (Master Degree of Science in Computer Science) – State University of Londrina, Londrina, 2016.

ABSTRACT

Embedded systems are present in various branches of human activity, to the point of up to 98% of the produced processors in world have been allocated in applications of this type in 2002. Given this wide applicability of embedded systems, it is common to find them in critical systems or equipment, making the testing phase during the development process of crucial importance. However, today the software testing activities are not very widespread and used within a large portion of the software development companies, in part because it is a time consuming, complex and costly step. To these factors it is also added up the lack of test documentation, problems from the human variable in the process and the need to perform regression tests after modifications, fixes or product updates. Faced with this situation, many efforts have been directed to researches related to test automation as a mean to reduce such problems. However there has been a challenge in building these tools that may require high development efforts, so that in many cases companies opt for manual testing. This scenario is common in software development since there are many different tests that must be applied during the product creation time plus there is a very wide range of components to be tested. Therefore, this proposal seeks for new development paradigms, called Model Driven Development, also known as MDD. The MDD is a concept that has been worked by the OMG (Object Management Group) from which this institution has proposed the MDA (Model Driven Architecture) in an approach that aims to raise the level of abstraction in the software development process. Against this backdrop, this work stablished a process of test automation system's development for embedded systems oriented by the architectures concepts of the MDA initiative. It was also accomplished an experimental test of this process and the analysis of the results, compared to other solutions found in the literature.

Keywords: Embedded Systems. Software Testing. Testing Automation. MDA.

LISTA DE ILUSTRAÇÕES

Figura 1 - Arquitetura de metamodelagem proposta pela OMG 43
Figura 2 - Relação entre modelos e metamodelos 44
Figura 3 - Relação entre elementos das camadas M0 e M1 45
Figura 4 - Relação entre elementos do metamodelo UML com os elementos do modelo da camada M1. 46
Figura 5 - Relação entre elementos das camadas M2 e M3 46
Figura 6 - Diagramas estruturais e comportamentais da UML 47
Figura 7 - Principais elementos construtivos do diagrama de classes da UML. 51
Figura 8 - Principais elementos construtivos do diagrama de sequência da UML.......... 52
Figura 9 - Relação entre temas quantidade de artigos por ano de publicação. 63
Figura 10 - Processo proposto para desevolvimento de sistemas de automação de testes funcionais. 64
Figura 11 - Processo de desenvolvimento proposto. 67
Figura 12 - Estrutura inicial para formalização da sintaxe BNF. 73
Figura 13 - Estrutura inicial do diagrama de classes. 79
Figura 14 - Criação de uma *Lifeline controller* do tipo *TestSuit*. 84
Figura 15 - Invocação do método *run()* do objeto controller da classe *TesteCase* pela classe *Testsuit* 85
Figura 16 - Criação da *Lifeline controller* da classe de TesteCase.................................... 86
Figura 17 - Acréscimo das *Lifelines* instanciadas ao digrama.. 87
Figura 18 - Esquema completo de diagrama de sequência para um caso de testes 88
Figura 19 - Acréscimo da classe Light por meio de especialização da classe ComponentOut 104
Figura 20 - Acréscimo das classes LuminositySensor e Switch por meio de especialização da classe ComponentIn 105
Figura 21 - Acréscimo da classe Camera por meio de especialização da classe ComponentCap. 106
Figura 22 - Modelo PIM completo do diagrama de classes do sistema 107
Figura 23 - Diagrama de sequência referente aos primeiros passos do caso de teste......... 108
Figura 24 - Diagrama de sequência diferenciando as ações de *setup* e do primeiro passo do caso de teste 110
Figura 25 - Modelo PSM do diagrama de classes 111

Figura 26 - Código estrutural gerado automaticamente pela ferramenta EA por meio de transformações MDA 112
Figura 27 - Trecho do diagrama de sequência submetido a mapeamento de elementos via documente XMI 113
Figura 28 - Linha de código em linguagem Java referente a instânciação de um objeto.. 114
Figura 29 - Organização física do sistema de automação de testes funcionais para um *cluster* automotivo 116
Figura 30 - Espia indicativa de farol acionada via entradas do SUT................................. 116
Figura 31 - Processamento digital de imagem aplicado à imagem capturada pelo sistema por meio da camera digital.. 116

LISTA DE TABELAS

Tabela 1- 10 princípios de teste de software apontados por Myers. 33
Tabela 2- Resultado e classificação dos trabalhos analisados.. 58
Tabela 3- Formulário de dados de teste.. 96
Tabela 4- Formulário de dados com a classificação de componentes..................................... 97

LISTA DE ABREVIATURAS E SIGLAS

MDA	*Model Driven Architecture*
OMG	*Object Management Group*
UML	*Unified Modeling Language*
MOF	*Meta Object Facility*
MDD	*Model Driven Development*
XMI	XML *Metadata Interchange*
ECU	*Electronic Control Unit*
OEM	*Original Equipment Manufacturers*
SUT	*System Under Test*
HIL	*Hardware In The Loop*
XML	*Extensible Markup Language*
CIM	*Computation Independent Model*
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
CWM	*Common Warehouse Metamodel*
OMT	*Object Modeling Technique*
OOSE	*Object-Oriented Software Engineering*
IEEE	*Institute of Electrical and Electronic Engineers*
IDE	*Integrated Development Environment*
ACM	*Association for Computing Machinery*
MDT	*Model Driven Testing*
PIT	*Platform Independent Test*
TTCN-3	*Testing and Test Control Notation Version* 3
BIT	*Built in contract testing*
URDAD	*Use Case, Responsibility Driven Analysis and Design*
EJB	*Enterprise Java Beans*
MDE	*Model Driven Engineering*
TPT	*Time Partition Test*
PDI	*Processamento Digital de Imagens*
AUTOSAR	*Automotive Open System Architecture*

BNF	*Backus-Naur Form*
mA	Miliampére
EA	*Enterprise Architect*
SysML	*Systems Modeling Language*
BPMN	*Business Process Modeling Notation*
BPEL	*Business Process Execution Language*
SoaML	*Service Oriented Architecture Modeling Language*
SPEM	*Software Process Engineering Metamodel*
WSDL	*Web Services Description Language*
QVT	*Queries/Views/Transform*

SUMÁRIO

1 INTRODUÇÃO 27
1.1 Objetivos 29
1.2 Organização do trabalho 30

2 REVISÃO BIBLIOGRÁFICA 31
2.1 Sistemas embarcados 32
2.2 Teste de software 33
2.3 Documentação e artefatos de teste 35
2.4 Automação de testes 36
2.5 *Model driven architecture* – MDA 39
2.6 Modelos no MDA 41
2.7 Transformações e mapeamentos do MDA 41
2.8 *Meta Object Facility* - MOF 43
2.9 Unified Model Language – UML 46
3.8.1 Diagrama de Classes 49
3.8.2 Diagrama de Sequência 51
3.2 RELAÇÃO MOF x UML 53
3.3 Ferramentas de MDA 54
3.4 Trabalhos Relacionados 55
3.4.1 Automação de testes em sistemas embarcados 55
3.4.2 Mapeamento Sistemático 57

4 PROPOSTA DE UM MDA PARA DESENVOLVIMENTO DE SISTEMAS DE AUTOMAÇÃO DE TESTES 64
4.1 4.1 Modelo independente de Computação – CIM 68
4.1.1 Análise do documento de testes 68
4.1.2 Classificação dos objetos 71
4.1.3 Definição de uma sintaxe para casos de teste 72
4.1.4 Transcrição dos casos de teste 77
4.2 Modelo Independente de Plataforma – PIM 77
4.2.1 Criação do Diagrama de Classes 78
4.2.1.1 Definição das classes Component Out 80

4.2.1.2 Definição das classes Component In 80
4.2.1.3 Definição das classes Component Cap 81
4.2.1.4 Definição das classes TestCase 82
4.2.2 Diagramas de Sequência 82
4.2.2.1 Mapeamento e transformação de CIM para PIM 83
4.3 Modelo Específico de Plataforma – PSM 89
4.3.1 Aplicação de transformações MDA automatizadas nos diagramas de Classes e Sequências 89
4.4 Código 90
4.4.1 Transformação de PSM para Código 90
4.4.2 Implementação de código comportamental 91
4.4.3 Compilação e execução do programa 91
4.5 Considerações do Capítulo 91

5 TESTE EXPERIMENTAL DA PROPOSTA 93
5.1 Materiais e Métodos 93
5.2 Teste Experimental 95
5.2.1 Modelo Independente de Computação – CIM 95
5.2.1.1 Análise do documento de testes 95
5.2.1.2 Classificação dos objetos 97
5.2.1.3 Definição de uma sintaxe para os casos de teste 98
5.2.1.4 Transcrição dos casos de teste 100
5.3 Modelo Independente de Plataforma 103
5.3.1 Criação do diagrama de classes 103
5.3.1.1 Definição das classes ComponentOut 103
5.3.1.2 Definição das classes ComponentIn 104
5.3.1.3 Definição das classes ComponentCap 105
5.3.1.4 Definição das classes TestCase 106
5.3.2 Diagramas de sequência 107
5.4 Modelo Específico de Plataforma – PSM 110
5.4.1 Aplicação de transformações MDA automatizadas nos diagramas de classe e sequência 110
5.5 Código 112
5.5.1 Transformação de PSM para Código 112

5.5.2 Execução 115
5.6 Considerações do Capítulo 117

6 CONCLUSÃO 119
6.1 Trabalhos futuros 122

REFERÊNCIAS 124

ANEXO I 133
ANEXO II 134
TRABALHOS PUBLICADOS PELO AUTOR 136

1 INTRODUÇÃO

A indústria de manufatura demanda uma elevada dinâmica de lançamentos de novos produtos com características e funcionalidades diferenciadas em função da pressão exercida pelo mercado [1]. Uma das formas adotadas para atender tais exigências é alcançada por meio de aplicação da eletrônica e computação embarcada nos mais variados produtos e ramos de atividade humana [2] [3].

No entanto, a despeito desses fatos, problemas nestes componentes podem acarretar em elevados custos decorrentes de *recalls*, danos à imagem da empresa ou ainda em casos extremos ocasionar perdas irreparáveis aos seus usuários. Dessa forma, a realização de testes apresenta-se como uma importante etapa do processo de desenvolvimento dos softwares embarcados nesses produtos. Todavia as atividades de teste caracterizam-se por ser uma etapa demorada, custosa e repetitiva. Polo [4] afirma que as atividades de teste podem representar até 60% de todos os esforços e custos de um projeto de desenvolvimento de software.

Dentro deste contexto, de acordo com Polo [4], Fewster e Graham [5] e Binder [6] a execução automática de testes é um fator que pode contribuir significativamente para economia de tempo e demais recursos quando comparada a processos manuais, uma vez que possibilita identificação de defeitos, agilidade no processo de depuração, captura e análise de resultados de forma consistente além de facilitar a execução de testes de regressão ao longo do processo de desenvolvimento. Não obstante, apesar de todas as vantagens apontadas pelos autores, atualmente essa prática ainda é muito pouco difundida dentro das empresas de desenvolvimento de Software [7]. Dentre os fatores que colaboram para essa situação estão o fato de que o desenvolvimento de um programa eficaz de automação de testes envolve um ciclo de desenvolvimento, que em muitos casos implica em elevado custo de implementação e manutenção desses sistemas de automação [7] [8].

Diante desse cenário, no qual o ciclo de desenvolvimento de sistemas de automação de testes pode acarretar em esforços e custos até mesmo maiores do que a realização manual dos testes, neste trabalho buscou-se então abordar essa problemática sob o enfoque do paradigma de desenvolvimento orientado por modelos, o *Model Driven Development* – MDD.

O MDD é um conjunto de metodologias onde o desenvolvimento é alicerçado no uso de modelos como documentos primários para o desenvolvimento de software [9], apresentando diretrizes e métodos para proporcionar um aumento no nível de

abstração das especificações de um software e também elevando o nível de automação durante o processo [10].

O MDA, por sua vez é uma implementação de MDD proposta e mantida pela OMG. De acordo com a OMG, o MDA deve ser entendido, não como um padrão único, mas sim como uma abordagem que contém uma coleção de padrões utilizados de forma conjunta, no qual se evidenciam os padrões e especificações da UML – *Unified Model Language* e o MOF – *Meta-Object Facility*, ambas definidas e mantidas pela OMG [11].

O MDA é uma abordagem de desenvolvimento que tem como desígnio fundamental, a criação de modelos como classe principal de artefatos para o desenvolvimento do software, utilizando ainda abordagens de transformações e geração automática de código a partir dos modelos. Essa abordagem possibilita ganhos como maior produtividade uma vez que automatiza a geração de códigos a partir dos modelos; portabilidade entre plataformas visto que o MDA propõe que o desenvolvimento seja pautado em modelos independentes de plataforma, podendo os mesmos serem transformados para modelos específicos ou código fonte e consistência entre código e conceitos/modelos dentre outras vantagens [11] [12] [13].

Considerando o foco na criação de modelos proposto no MDA, a OMG estabelece e faz uso de três tipos de abstrações de modelos para essa proposta, sendo elas representadas pelas siglas CIM (*Computation Indepent Model*), PIM (*Platform Indepent Model*) e PSM (*Platform Specific Model*). Estas tipologias de modelos são empregadas no MDA com o propósito de descrever um sistema de software sob uma perspectiva particular. O CIM fornece uma visão independente de conceitos computacionais, focando principalmente papel do sistema no ambiente em que o mesmo irá operar [14]. O PIM por conseguinte é um modelo que enfatiza a descrição da operação do sistema ao mesmo tempo em que oculta características de uma plataforma particular [14]. Por fim o PSM apresenta uma visão do sistema que considera detalhes específicos a uma plataforma de desenvolvimento em particular [14].

Complementarmente a essas três visões de modelos estipuladas pela OMG, o MDA possui ainda como intuito a transformação entre modelos. Ou seja, o MDA prega a transformação entre modelos da diferentes visões apresentadas a partir de mapeamentos responsáveis por correlacionar os elementos presentes em um modelo fonte com os elementos de um modelo alvo [11].

1.1 Objetivos

O objetivo geral deste trabalho é a criação de processo de desenvolvimento de sistemas de automação de testes para o domínio específico de dispositivos com sistemas embarcados a partir de um conjunto de modelos especializados para o referido domínio. Os modelos foram desenvolvidos seguindo as diretrizes do MDA orientados pela arquitetura dos padrões UML, XMI e MOF.

Nesse sentido, foram norteados e estabelecidos os seguintes objetivos específicos, a saber:

1. Estudar e compreender a aplicação do MDA para um domínio específico de aplicação: Sistemas de automação de teste para sistemas embarcados;
2. Propor um processo orientado e amparado pelas diretrizes do MDA e suas transformações de modelos CIM para PIM, PIM para PSM e PSM para código para o referido domínio;
    2.1. Definição de sintaxe para normatização de documentos de teste para viabilizar a transformação MDA de CIM para PIM;
3. Realizar a análise da proposta por meio de um teste experimental com base em:
    1. Avaliar a aplicabilidade da proposta de transformações MDA para a criação de sistemas no contexto da automação de testes de software para sistemas embarcados;
    2. Criar um processo que institua um formalismo que possa guiar o desenvolvimento de sistemas e arquiteturas de teste de software e que seja pautado por padrões amplamente difundidos e utilizados na área, como os adotados e mantidos pela OMG;
    3. Possibilitar a criação de arquiteturas de teste baseadas em scripts a partir da utilização de modelos e possibilitando através do MDA a transformação destes para qualquer linguagem de programação desejada;

Ressalta-se por fim, que não é objetivo deste trabalho investigar de forma individual e em profundidade cada uma das linhas de pesquisa, ferramentas de testes automatizados e aplicações baseadas no MDA, mas sim buscou-se combinar as técnicas já estabelecidas em um processo novo, o MDATT – *Model Driven Architecture for Test Tools*, de maneira a agregar os benefícios dessa abordagem ao desenvolvimento de Sistemas de Testes Automatizado.

1.2 Organização do trabalho

Os demais capítulos deste trabalho estão organizado e distribuídos da seguinte forma:

O capítulo 2 discorre sobre os conceitos pertinentes à problemática abordada neste trabalho, de forma a proporcionar o embasamento teórico sobre o assunto. A princípio são abordados os conceitos acerca dos sistemas embarcados, teste e automação de testes de software com o intuito de caracterizar a problemática em estudo. Além disso, são apresentadas também visões gerais sobre o MDA, assim como outros temas relacionados como o MOF e UML, buscando-se assim melhor compreender a estrutura e organização desses padrões, de forma a proporcionar uma visão mais holística sobre os mesmo e como aplica-los à problemática em estudo. Por fim, apresenta-se ainda nesse capítulo uma seção de trabalhos relacionados, a partir de um mapeamento sistemático da literatura sobre o tema.

A proposta de processo de desenvolvimento de sistema de automação de testes funcionais embarcados baseado no MDA é apresentada no capítulo 3. Este capítulo descreve em detalhes todas as atividades e artefatos realizadas e geradas ao longo do processo.

O capítulo 4 apresenta a proposição de teste experimental para avaliação do processo, realizado a partir de um caso de testes baseado na indústria automotiva.

Por fim, no capítulo 5, são apresentadas as conclusões e análise de resultados deste trabalho.

2 REVISÃO BIBLIOGRÁFICA

A automação de parte dos testes de um projeto é apontada por diversos autores [7] [4] como uma solução para a redução de tempo e custos de produção, de forma que diversas tecnologias e ferramentas para automação de testes foram propostas e desenvolvidas, como as tecnologias de *capture and replay*, XUnit *frameworks* e ferramentas de automação de combinação de dados para testes [4]. Entretanto, faz-se relevante ressaltar que a automação de testes é uma etapa importante no desenvolvimento de um sistema de software, requerendo assim, um ciclo de desenvolvimento próprio que deve abranger desde etapas de planejamento, modelagem, codificação e execução dos testes automatizados [8] [15].

Diante desse cenário, introduz-se os conceitos de *Model Driven Development* – MDD e *Model Driven Architecture* - MDA que é uma implementação do MDD proposta pela OMG que enfatiza essencialmente as transformações entre modelos, sendo fundamentalmente amparada pelas tecnologias e padrões mantidos pela OMG como MOF, UML e XMI. Frente a esses novos paradigmas e diretrizes de desenvolvimento propostos pelo MDD e MDA, este trabalho propõe um processo de desenvolvimento baseado no MDA voltado para a criação de ferramentas de automação de testes funcionais para sistemas embarcados.

Dessa forma, foi então realizado um levantamento bibliográfico com o intuito de estudar os principais conceitos e áreas de conhecimento para o embasamento e desenvolvimento da proposta, assim como foram levantados e estudados outros trabalhos relacionados com o tema.

À vista das informações apresentadas, o restante deste capítulo abordará inicialmente os conceitos relacionados com o domínio da aplicação, no caso os sistemas embarcados, na sequência serão tratados os conceitos a respeito de processos e métodos de desenvolvimento necessários ao desenvolvimento desses produtos, passando então para o estudo dos testes de software assim como meios para a automação dos mesmos, finalizando com o estudo dos conceitos a respeito do MDA e trabalhos relacionados ao tema.

2.1 Sistemas embarcados

De acordo com Heath [16] os sistemas embarcados são definidos como sistemas desenvolvidos para controlar uma função ou um conjunto de funções, baseados em microprocessadores, e que não são projetados para serem programados como os computadores pessoais. Segundo Moreira [17] durante o ano de 2002, somente 2% dos microprocessadores fabricados no mundo foram destinados para aplicações em computadores pessoais, enquanto que os 98% restantes foram alocados em aplicações embarcadas diversas, que englobam desde brinquedos, telefones celulares a até aplicações aeroespaciais.

Um ramo de atividade industrial que faz uso frequente de sistemas embarcados em seus produtos é o do setor automotivo. Um exemplo de aplicação desses sistemas são as ECU's (*Electronic Control Unit*) que são módulos instalados em locais diversos em um automóvel, sendo responsáveis por inúmeras tarefas como, controle de injeção eletrônica, controle de tração e estabilidade, rastreamento, entre outros. De acordo com Broy et. al [18], estima-se que atualmente, um automóvel possui em torno de algumas dezenas de milhares de linhas de codificação embarcada.

Em função de sua vasta aplicabilidade em variados ramos da atividade humana, o processo de desenvolvimento de softwares voltados para o controle de sistemas embarcados possui diversas características que diferem ao processo de desenvolvimento de outros tipos de software de propósito geral. Pinheiro [19], Carro e Wagner [20], apontam algumas limitações e necessidades inerentes ao processo de desenvolvimento de sistemas embarcados como:

a) capacidade de processamento e espaço em memória limitados;

b) Necessidade de redução de consumo de energia;

c) elevado grau de confiabilidade requerido por aplicações desse tipo;

d) escolha de componentes de forma a otimizar a aplicação desenvolvida, dentre outras.

Wolf [21] acrescenta que os sistemas embarcados em muitos casos são concebidos para que possam manter-se em funcionamento de maneira independente durante toda a sua vida útil, sem a necessidade de intervenções.

Dessa forma, o desenvolvimento de sistemas embarcados automotivos é uma tarefa bastante complexa em função principalmente das limitações citadas assim como do elevado grau de exigência das montadoras e OEM’s (*Original Equipment Manufacturers*)

em termos de qualidade e confiabilidade de seus produtos, advindas da, cada vez maior exigência por conforto, segurança e qualidade por parte de seu público consumidor.

Inserido dentro desse processo de desenvolvimento está o desenvolvimento e validação dos softwares embarcados. Diante deste contexto de elevado nível de qualidade e confiabilidade impostos pela indústria é imprescindível a realização criteriosa de testes durante o processo de desenvolvimento.

2.2 Teste de software

De acordo com Pressman [22] a atividade de teste consiste em uma execução controlada de um software com a finalidade de encontrar um erro, sendo assim um elemento crítico para garantia de qualidade de software e representando ainda a revisão final da especificação, projeto e geração de código. Sommerville [23] acrescenta ainda que testar, significa exercitar um programa usando dados reais que deverão ser processados pelo programa e por fim, fazer uma análise detalhada dos resultados ou saídas apresentadas, em busca de anomalias comportamentais.

Segundo Myers [24] a fase de teste de Software é uma das etapas mais custosas dentro de um projeto de desenvolvimento de Software, sendo as atividades relacionadas a testes responsáveis por consumir até 50% de todo o tempo dedicado ao projeto e mais de 60% do custo total do mesmo. Entretanto, mesmo demandando tamanha quantidade de recursos, é fato que a realização de testes exaustivos na busca por detectar todos os erros em um programa é algo inviável, ou mesmo impossível na maioria dos casos [24].

Dessa forma, algumas estratégias de teste e princípios fundamentais para realização de testes são propostas por Myers. A tabela 1 apresenta uma listagem dos 10 princípios de Teste de Software considerados como vitais e que atuam como guias norteadores para a criação e execução de testes [24].

Tabela 1: 10 princípios de teste de software apontados por Myers. [24]

1º Princípio	Uma parte importante de um caso de teste é a definição de

	resultado esperado.
2º Princípio	Um programador deve evitar testar seu próprio programa.
3º Princípio	Os casos de teste não devem ser grandes e exaustivos.
4º Princípio	Analisar abundantemente os resultados de cada teste.
5º Princípio	Casos de teste devem ser escritos com entradas que são inválidas e não esperadas, como também com entradas válidas e esperadas.
6º Princípio	Examinar um programa para ver se ele não faz que é proposto, é apenas metade da batalha; a outra metade é verificar se o programa faz o que não é proposto.
7º Princípio	Casos de teste descartáveis devem ser evitados, a menos que o programa seja também descartável.
8º Princípio	Nunca se deve assumir que nenhum erro vai ser encontrado.
9º Princípio	A probabilidade de encontrar mais erros numa seção de um programa é proporcional ao número de erros já encontrados naquela seção.
10º Princípio	Testar é uma tarefa extremamente criativa e intelectualmente desafiante.

Complementarmente aos 10 princípios listados, duas das estratégias mais usuais para a realização de testes são os chamados *Black-Box Testing*, ou testes de caixa preta e *White-Box Testing* ou teste de caixa branca. Ambos os conceitos são abordados abaixo de acordo com Pressman [22]:

- Testes de caixa branca, também conhecidos como testes estruturais que visam um exame detalhado da estrutura do software, buscando verificar se as condições lógicas e laços internos do software não apresentam erros. Geralmente, essa estratégia de teste busca garantir que todos os caminhos independentes de um dado módulo sejam exercitados ao menos uma vez,

assim como todas as decisões lógicas e estruturas de dados internas ao mesmo.

- Testes de caixa preta, também são conhecidos como testes funcionais pois visam somente verificar se os requisitos funcionais do software são satisfeitos, de forma que a partir de uma dada entrada, as saídas esperadas são geradas, não existindo preocupação quanto à estrutura lógica interna do software. Ao contrário dos testes de caixa-branca que geralmente são realizados no início do processo de testes, os testes de caixa-preta tendem a ser aplicados nas fases finais do projeto. Este trabalho aborda exclusivamente os testes funcionais.

2.2.1 Documentação e artefatos de teste

Dentro de um processo de criação e execução de testes em um produto de software, alguns artefatos e subprodutos podem ser gerados, geralmente na forma de documentos de especificação de casos de testes ou ainda como relatórios de resultados da execução dos testes.

Nesse sentido, a norma IEEE 829 – IEEE *Standard for software test documentation* foi desenvolvida com o intuito de apresentar e descrever um conjunto de documentos de teste que visam auxiliar no processo de desenvolvimento e organização dos testes de software. Essa norma especifica a forma e conteúdo de oito tipos de documentos relacionados às diversas etapas de um processo de testes. Entretanto, essa norma não especifica quais documentos devem ser utilizados, cabendo à equipe ou responsável pelo processo a escolha de quais documentos adotar [25].

Os documentos especificados nessa norma são:

- Plano de Testes: Neste documento são definidos o escopo, abordagem, recursos e cronograma das atividades de teste. São identificados ainda, os itens a serem testados, com foco nos recursos, pessoas, e riscos associados ao plano;
- Especificação de projeto de teste: Este documento representa um refinamento do plano de testes onde são identificadas as características a serem testadas assim como definidos critérios de aprovação/reprovação dos testes;

- Especificação de casos de teste: Neste documento são especificados os casos de teste a serem executados explicitando os dados de entrada e saída esperada para cada passo de teste, assim como as ações necessárias e ainda equipamentos necessários para reconhecimento de condições ambientais.
- Especificação de procedimento de teste: Este documento especifica a sequência de passos necessários à execução de um caso de teste, além de especificar requisitos ou necessidades especiais para a execução dos passos;
- Relatório de Encaminhamento de Item de Teste: Este relatório apresenta os itens de teste encaminhados para a execução, contendo as especificações de status e local onde será realizado o teste;
- Diário de Teste: Este documento busca prover um registro histórico de ações e ocorrências relevantes durante a execução dos casos de teste;
- Relatório de incidentes: Neste documento são relatados todo e qualquer evento ocorrido durante o processo de teste que necessite de uma avaliação ou investigação;
- Relatório de Resumo de Teste: Este documento ordena um resumo dos resultados das atividades testes e contempla ainda as avaliações dos testes baseadas nesses resultados;

No entanto, apesar dessa norma ser bem difundida, sua aplicação está aquém de sua plenitude dentro das empresas de desenvolvimento de software. É comum que empresas de desenvolvimento adotem parcialmente as especificações da norma IEEE 892, ou mesmo não façam uso dela, criando a documentação de testes de formas informais ou seguindo um padrão interno da empresa [7].

2.3 Automação de testes

As empresas e desenvolvedores de software frequentemente veem-se face a uma realidade de cobranças cada vez maiores por produtos de maior qualidade e confiabilidade e que ainda atendam a uma demanda de *time-to-market* cada vez mais baixa. Dustin [8], descreve tal cenário onde:

> “[...] os gerentes e desenvolvedores de Software são cobrados a disponibilizar seus produtos dentro de prazos bastante curtos, e ainda com recursos limitados. Mais de 90% dos desenvolvedores já, em algum momento extrapolaram datas de entrega. Estourar os prazos é uma situação recorrente para 67% dos desenvolvedores. Somado a isso, 91% dos desenvolvedores já se viram forçados a remover funcionalidades chave do produto em um momento avançado do ciclo de desenvolvimento com o intuito de atender aos prazos. [8]”

Conseguir inserir um produto no mercado, tão rápido quanto possível, pode em alguns casos realmente ser o diferencial entre a sobrevivência ou morte do produto em questão, e em casos mais drásticos, a sobrevivência ou morte da empresa em si [8]. Ramler e Wolfmaier [26] afirmam que esse cenário de crescente competitividade internacional associada a prazos de desenvolvimento apertados, etapa usualmente extensa e dispendiosa atrelada diretamente ao ciclo de desenvolvimento. Alternativamente, e no intento de contornar esse panorama muitas empresas têm focado seus esforços na completude de testes no menor tempo possível, levando à automação dos mesmos [26]. Dustin [8] define a automação de testes como sendo a automação do gerenciamento e execução de atividades de teste, incluindo o desenvolvimento e execução de scripts de teste e a verificação dos requisitos de teste por meio de ferramentas automatizadas. Thummalapenta [27] acrescenta que a automação de testes é uma atividade em que se busca criar uma representação mecanicamente interpretável de um caso de teste manual. Além disso, segundo Kusarinen [7] as pesquisas realizadas no âmbito da Engenharia de Software possuem dois objetivos primários: a redução de custos de produção e o aumento da qualidade dos produtos de software. Uma vez que as atividades de teste de software representam um papel de grande importância para esses dois quesitos, a implementação adequada da automação nos processos de software pode possibilitar a combinação desses dois objetivos.

Além disso, é comum que um mesmo teste tenha que ser realizado inúmeras vezes durante o processo de desenvolvimento de forma que a automação de testes mostra-se vantajosa, uma vez que possibilita a repetição do mesmo em diferentes momentos do desenvolvimento [28]. Todavia, apesar da consonância acerca do fato de que a qualidade do software em produção pode ser melhorada pelo uso de processos automatizados de teste, presentemente essa prática apresenta escassez de difusão junto às empresas de desenvolvimento de software. Um dos obstáculos defrontados por essas empresas no momento da implantação da automação de testes está relacionado à ausência de conhecimentos sobre linguagens de programação específicas para a criação de scripts de

automação, ou ainda conhecimentos a respeito das ferramentas utilizadas para isso. Ou seja, a automação de teste requer o uso conjunto de habilidades e conhecimentos relacionados ao teste em si, e também de habilidades bastante específicas de programação [29].

Dessa forma, dentro desse contexto, Leitner [30] realiza um importante estudo comparativo entre técnicas manuais e automatizadas de teste, que gerou a constatação de que ambas as técnicas são complementares mutuamente. Ao passo que testes automatizados permitem uma maior cobertura de casos de testes com menor exigência de esforço dos desenvolvedores em termos de execução dos mesmos, os testes manuais possibilitam o desenvolvimento de casos de testes mais complexos com dados de entrada diversificados [31]. Em suma, os testes automatizados atuam pelo viés de abrangência, que proporciona uma cobertura mais ampla dos casos de teste, à medida que os testes manuais conseguem ser mais incisivos e capturam casos especiais e mais profundos. Entretanto, Rätzmann [15] ressalta que é impossível testar todos os aspectos de um software, mesmo com o uso intensivo de automação. Além disso, testes automatizados devem ser continuamente atualizados. Caetano [31] afirma que a escolha dos testes que serão automatizados deve ser realizada de acordo com o contexto de cada projeto. No entanto, apesar de não existir uma categorização padrão e amplamente difundida, o mesmo autor aponta alguns tipos de teste que se mostram mais suscetíveis à automação, como testes de regressão, testes em funcionalidades críticas de um software ou testes que envolvam tarefas longas ou repetitivas.

Dustin [8] afirma que um efetivo programa de teste, que incorpora a automação de teste de software, envolve por si só um miniciclo de desenvolvimento, abordando etapas comuns ao desenvolvimento de software de propósitos gerais como planejamento, definição e requisitos de teste, projeto, desenvolvimento, execução e avaliação das atividades. Hoffman [32] afirma que arquiteturas de automação de teste devem ser pensadas para cada caso e facilmente se tornam obsoletas mediante alterações ou atualizações no SUT – *System Under Test*. Em especial quando se trata da implementação de uma arquitetura de automação de testes voltada para sistemas embarcados, existem alguns fatores adicionais a serem analisados uma vez que tais sistemas interagem continuamente com o ambiente e são compostos não somente por um sistema de software, mas também por diversos outros componentes eletromecânicos [28]. Em consequência desse fato, a realização de testes nesses sistemas deve ser capaz de abranger essa interação software/*hardware*, o que levou ao desenvolvimento de técnicas como o *Hardware-in-the-Loop* - HIL, na qual os testes são realizados diretamente na versão física do *hardware* de um sistema embarcado.

2.4 *Model driven architecture* – MDA

O MDA é uma iniciativa que tem como principal objetivo, extrair valor a partir de modelos e processos de modelagem, proporcionando assim uma forma de lidar com a alta complexidade e interdependências existentes em sistemas de software. Dessa forma, o padrão MDA da OMG prega que existem diversas maneiras de agregar valor a modelos em um processo de desenvolvimento, como [11]:

- Utilização de modelos como meio de intercomunicação: Facilitar o entendimento comum entre os membros de uma equipe ou comunidade é uma das premissas básicas do processo de modelagem de um sistema. O padrão imposto pelo MDA busca auxiliar nessa tarefa de três formas: a) provendo um conjunto bem definido de termos, ícones e notações que auxiliam no entendimento comum em uma determinada área de conhecimento; b) provendo os fundamentos necessários para modelagem como forma de gerir, versionar e compartilhar dados semânticos e; c) provendo bibliotecas de modelos reutilizáveis como regras e vocabulários, processos e padrões arquiteturais de design. Os modelos se tornam assim, parte integrante da "memória corporativa" de uma empresa capturando os requisitos, processos, informações e serviços da mesma.
- Derivações automáticas de modelos via automação: Uma das propostas mais proeminentes do MDA é a derivação automatizada de artefatos ou implementações baseadas em modelos. A automação proposta pelo MDA visa reduzir tempo e custos para o desenvolvimento e manutenção de sistemas, além de proporcionar uma maior consistência ao processo de desenvolvimento como um todo. A derivação automática de artefatos de software envolve três elementos básicos, sendo eles, um modelo de entrada (*source model*), uma transformação e um modelo de saída (*target model*). Em muitos casos, o modelo de saída pode ser representado por outro modelo, em um nível de menor abstração.
- Simulação e execução de modelos: A visão do MDA sobre modelos como uma forma bem definida e formal de representação de dados possibilita que os modelos possam ser submetidos a "*Engines*" de simulação e execução,

proporcionando assim uma melhor compreensão sobre as funcionalidades de um sistema, assim como uma forma de validação dos próprios modelos.

- Extração de informações de modelos: Modelos criados a partir de uma estrutura bem definida e formalizada podem ser usados para a extração de informações contidas nos mesmos, assim como podem ser utilizados para a geração de outros modelos ou artefatos, como por exemplo, *playbooks* de processos, documentação, especificações de aquisição ou mesmo sistemas de software.

Todavia, salienta-se que a principal característica do MDA se concentra nas transformações entre modelos. A automação das transformações de modelos de alto-nível para sistemas de informação executáveis proporciona uma série de vantagens, a citar a redução de custos, tempo, risco de produção e manutenção de um sistema, ao mesmo tempo em que atua no refinamento do sistema para o domínio específico do problema [11]. Entretanto, nem todos os modelos são aptos a transformações automáticas. Um modelo precisa ser suficientemente completo e preciso de forma que as informações sobre o sistema em desenvolvimento estejam bem especificadas pelos modelos. Silva [33] acrescenta que:

> "O MDA exige a utilização de uma linguagem de modelagem bem definida, que possa ser interpretada por homens e computadores. Uma linguagem bem definida é aquela que obedece um conjunto de regras formais. O MOF, é uma linguagem para descrição de metamodelos, que por sua vez irão definir regras sobre como os modelos de uma aplicação devem ser escritos para serem considerados bem definidos." [33]

Dessa forma, o MDA propõe duas abordagens primárias para a automação de transformações evolvendo modelos [11]:

- Um padrão de transformação é aplicado a um modelo de forma a obter-se como resultado um modelo que seja específico para uma dada plataforma. Por exemplo, um modelo de negócios pode ser transformado para implementações das tecnologias XML *Schema* ou para um código em C# ou Java.
- Uma "*Engine*" para execução de modelos implementada em alguma plataforma, que diretamente executa um dado modelo e detenha a capacidade adicional de gerar e/ou consumir dados a partir do formato XML.

Em ambos os casos, o objetivo primário da proposta MDA é atingido, uma vez que as informações contidas em um modelo se tornam úteis no sentido de gerar um sistema executável ou ainda um sistema de simulação.

2.4.1 Modelos no MDA

Os modelos e as relações entre modelos são um dos pontos principais na abordagem de desenvolvimento MDA. Nessa abordagem, foram estabelecidas três categorias de modelos, de acordo com o ponto de vista adotado e a "intenção" do modelo, sendo eles:

Modelo Independente de Computação: O Modelo Independente de Computação ou *Computation Independent Model* (CIM) é uma visão do sistema em que a compreensão independe do conhecimento em computação. A modelagem de requisitos em um sistema geralmente é realizada usando-se um modelo independente de computação, focando o papel do sistema no ambiente em que o mesmo irá operar de forma a auxiliar o entendimento das atribuições do sistema [14].

Modelo Independente de Plataforma: O Modelo Independente de Plataforma ou *Platform Independent Model* (PIM), é uma visão do sistema independente da plataforma de implementação. O PIM foca simultaneamente na descrição da operação do sistema, e em ocultar características de uma plataforma particular, podendo assim ser reaproveitado em diferentes plataformas [14].

Modelo Específico de Plataforma: O Modelo Específico de Plataforma ou *Platform Specific Model* (PSM), é uma visão do sistema que considerará detalhes específicos à plataforma de desenvolvimento. Ele é criado a partir do Modelo Independente de Plataforma (PIM) [14]. Um PSM pode já ser considerado como uma implementação do sistema, caso forneça todas as informações necessárias para a construção e funcionamento do mesmo [14].

2.4.2 Transformações e mapeamentos do MDA

As transformações são atividades que visam gerar modelos alvo a partir de outros modelos fonte, de acordo com um mapeamento que descreve como um modelo fonte pode ser transformado em um modelo alvo [34].

Lopez et al [35] propõem a seguinte conceituação para os termos transformação e mapeamento:

- Mapeamento: o termo mapeamento é utilizado como sinônimo de correspondência entre elementos de dois metamodelos. Um mapeamento entre dois elementos define uma correspondência entre estes.
- Transformação é a atividade de transformar os elementos de um modelo fonte em elementos de um modelo alvo.

O MDA propõe três padrões diferentes de mapeamento entre modelos, sendo elas, de acordo com Chaves [36]:

- Mapeamento baseado em tipos de elementos: o modelo independente de plataforma deve ser criado a partir de uma linguagem que suporte um determinado conjunto de elementos de modelo, usualmente a UML. Os mapeamentos têm como função definir a correspondência existente entre o tipo de cada elemento presente no modelo independente de plataforma com um ou mais tipos de elementos no modelo específico de plataforma. Este tipo de mapeamento dificulta a influência de um arquiteto de software no processo de transformação, uma vez que os mapeamentos entre os tipos presentes no PIM e no PSM ocorrem de maneira estática.
- Mapeamento baseado em instâncias de elementos: nesta abordagem de mapeamento, para cada elemento do modelo independente de plataforma é definido qual o mapeamento que será utilizado pelo mesmo na transformação. Isso é feito através das anotações inseridas no PIM na forma de marcações que dirigem a transformação para o PSM. Esta forma de mapeamento faz uso de uma cópia do modelo PIM, na qual são efetivamente inseridas as marcações, com o intuito de preservar o modelo inicial em sua forma original.
- Mapeamento baseado em tipos e instâncias de elementos: esta abordagem é segundo Chaves [36], a forma mais interessante de mapeamento, uma vez que faz uso conjunto das duas técnicas anteriores. Esse tipo de mapeamento possibilita uma maior interação e influência do arquiteto ou engenheiro de software no processo de transformação de modelos, visto que mapeamentos pré-definidos baseados em tipo podem ser adaptados ou incrementados por meio da inserção de anotações ou marcações em elementos do modelo, conduzindo assim a transformação da maneira desejada.

2.8 Meta Object Facility - MOF

O MOF – *Meta Object Facility* é uma especificação estabelecida pela OMG com o intuito de criar um padrão comportamental que possa ditar formas de compartilhamento de modelos entre diversas ferramentas computacionais [37].

Essa especificação consiste de um metamodelo que descreve uma tecnologia neutra e que define uma linguagem abstrata para metamodelagem e um *framework* para especificação, construção e gerenciamento de metamodelos Orientado a Objetos [38]. Dessa forma, o MOF serve como base para diversas outras especificações estabelecidas por essa entidade, merecendo destaque a Linguagem de Modelagem Unificada – UML, o CWM - *Common Warehouse Metamodel*, o formato de compartilhamento de modelos XMI e em última instância o próprio MDA.

A arquitetura de metamodelagem definida pela OMG é baseada em uma estrutura hierárquica composta por quatro camadas, nomeadas de M0 a M3, sendo a camada M3 a mais elevada [38], conforme estrutura elucidada na Figura 1:

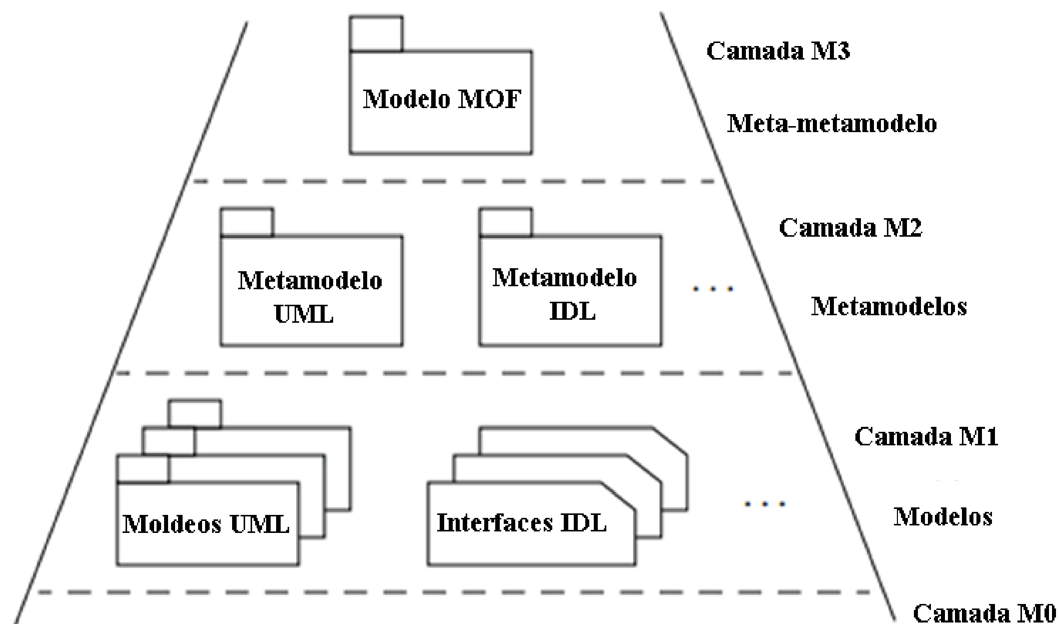

Figura 1: Arquitetura de metamodelagem proposta pela OMG. [38]

Algumas considerações importantes devem ser traçadas a respeito dessa arquitetura, quando analisada sob a perspectiva do MOF [38]:

- O modelo MOF é orientado a objetos e possui muitos elementos construtivos em conformidade/alinhamento com os elementos construtivos

da UML. Essa proximidade conceitual entre os dois metamodelos possibilita que o modelo MOF seja descrito a partir de elementos da própria UML.

- Os meta-níveis descritos pela Figura 1, não são fixos e imutáveis. A OMG propõe tal arquitetura, entretanto, não estabelece restrições à alteração da mesma. Assim, os meta níveis da arquitetura proposta pela OMG, podem ser entendidos como um convenção para facilitar o entendimento das relações existentes entre diferentes tipos de dados e metadados.
- Um modelo baseado em MOF, não precisa necessariamente estar limitado a uma única camada da arquitetura, podendo, de acordo com as necessidades, "flutuar" entre as camadas ou mesmo ocupar mais de uma delas.
- O modelo MOF é alto-descritivo, ou seja, sua estrutura é descrita através de elementos especificados em seu próprio metamodelo.

A arquitetura de metamodelagem da Figura 1 estabelece uma relação entre os diversos níveis existentes utilizando os conceitos de modelos, linguagem, metamodelos e metalinguagens. Sebesta [39] define uma metalinguagem como a linguagem utilizada para descrever outras linguagens. Uma metalinguagem é descrita por um meta-metamodelo, que por sua vez é descrito por uma meta-metalinguagem [39]. A Figura 2 exemplifica esse relacionamento.

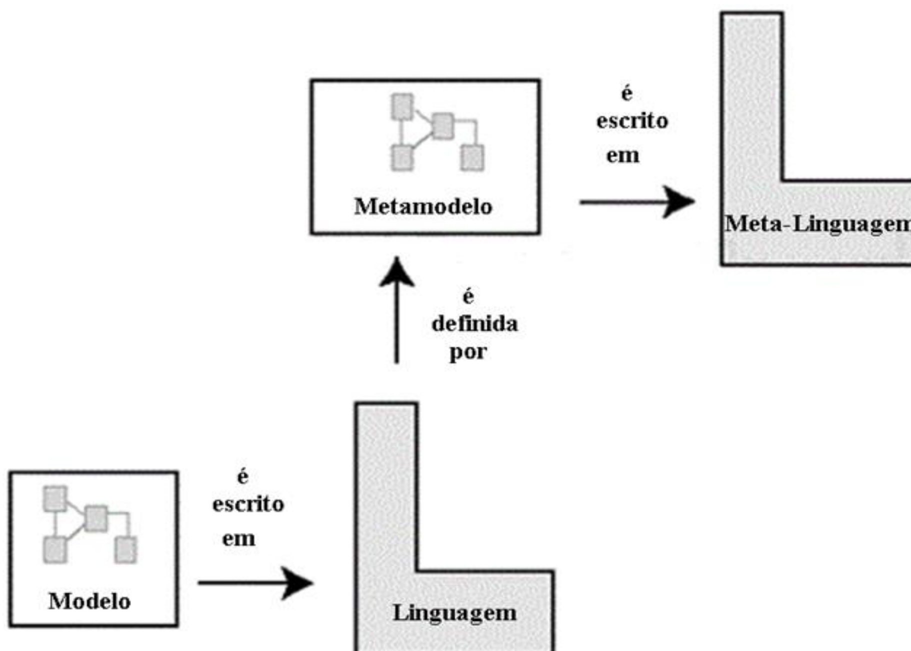

Figura 2: Relação entre modelos e metamodelos. [34]

Vale ressaltar a relevância de definir qual o escopo de cada uma das camadas pertencentes à arquitetura de metamodelagem da OMG [38].

Camada M0: Essa é a camada de informação da arquitetura, sendo composta principalmente de objetos e dados em tempo de execução. Ou seja, as instâncias reais que compõem um sistema de software encontram-se acomodadas nessa camada.

Camada M1: Essa Camada define os modelos do usuário que são a base para as instâncias presentes na camada imediatamente abaixo. Ou seja, os elementos residentes nessa camada, são classificações ou ainda categorizações em um nível mais elevado de abstração das instâncias presentes na camada M0. A Figura 3 exemplifica essa relação entre elementos das duas camadas.

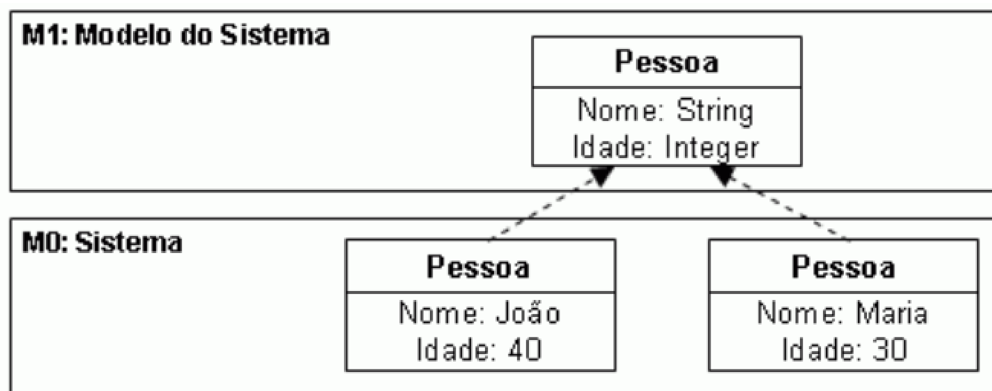

Figura 3: Relação entre elementos das camadas M0 e M1 [34].

Camada M2: Essa camada contém o metamodelo que descreve a linguagem de modelagem utilizada para elaborar o modelo presente na camada M1. Isto é, a relação existente entre metamodelo e modelo é análoga à existente entre modelo e suas instâncias. O metamodelo especifica como categorizar cada elemento existente no modelo da camada M1. A Figura 4 representa a relação existente entre elementos do metamodelo da UML, com seus respectivos elementos da camada M1.

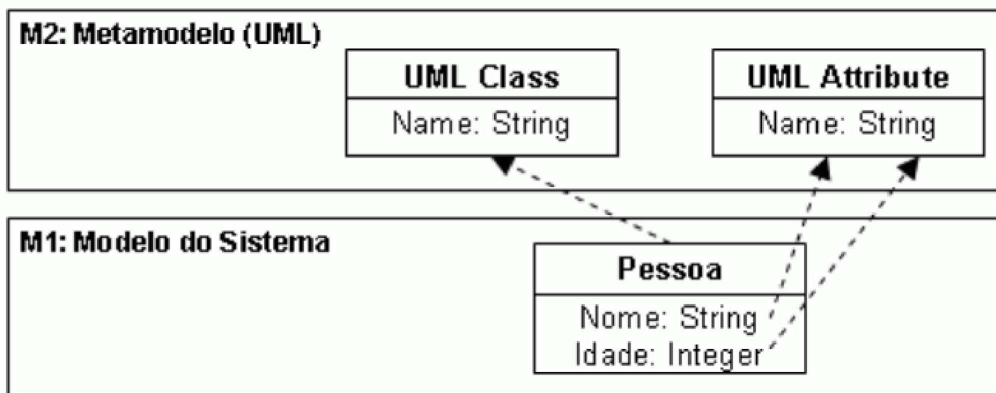

Figura 4: Relação entre elementos do metamodelo UML com os elementos do modelo da camada M1 [34].

Camada M3: Essa camada compreende o meta-metamodelo que define a linguagem utilizada para definição dos metamodelos presentes na camada M2. É nesse nível hierárquico em que se encontra a especificação do MOF. A mesma relação existente entre os elementos das camadas M0 e M1 e as camadas M1 e M2, também está presente nas camadas M3 e M2. Dessa forma, os elementos especificados pelo MOF na camada M3, classificam os elementos do metamodelo presente na camada M2. A Figura 5 mostra essa relação.

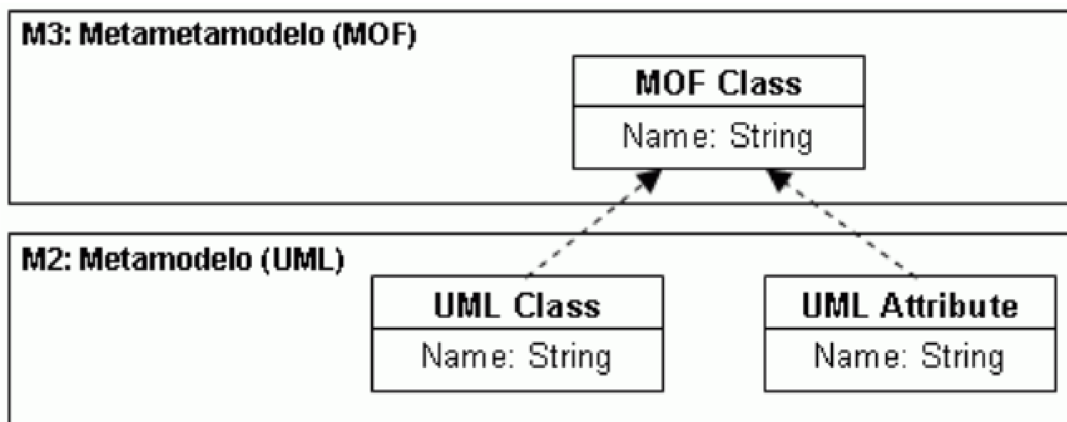

Figura 5: Relação entre elementos das camadas M2 e M3 [34].

2.9 Unified Model Language - UML

A UML é uma linguagem gráfica para a modelagem de software orientado a objetos. Ela é resultante da união de três formatos de modelagem: o método de Booch, o método OMT (*Object Modeling Technique*) de Jacobson e o método OOSE (*Object-Oriented Software Engineering*) de Rumbaugh [40].

Posteriormente à junção, várias empresas da área de modelagem e desenvolvimento de software passaram a contribuir para o projeto com o propósito de melhorar e ampliar a linguagem [40]. Assim, em 1997 a UML foi adotada pela OMG como uma linguagem padrão de modelagem. Segundo Pressman, a compreensão do vocabulário da UML (os elementos visuais e seus significados) permite o entendimento e a especificação de um sistema e a fácil comunicação das características de um projeto de sistema para outros interessados. A UML apresenta como características principais [22]:

- É independente do domínio da aplicação;
- É independente do processo ou metodologia de desenvolvimento;
- É independente das ferramentas de modelagem;
- Apresenta mecanismos poderosos de extensão;
- Agrega um conjunto significativo de diagramas e técnicas.

Os diagramas que compõem esse formato de modelagem são: diagrama de casos de uso, diagrama de classes, diagrama de objetos, diagrama de pacotes, diagrama de sequência, diagrama de comunicação, diagrama de máquina de estados, diagrama de atividade, diagrama de visão geral de interação (a partir da UML 2.0), diagrama de componentes, diagrama de implantação, diagrama de estrutura composta e diagrama de tempo ou de temporização [40]. Estes podem ser divididos em dois tipos: estruturais e comportamentais, de acordo com a Figura 6.

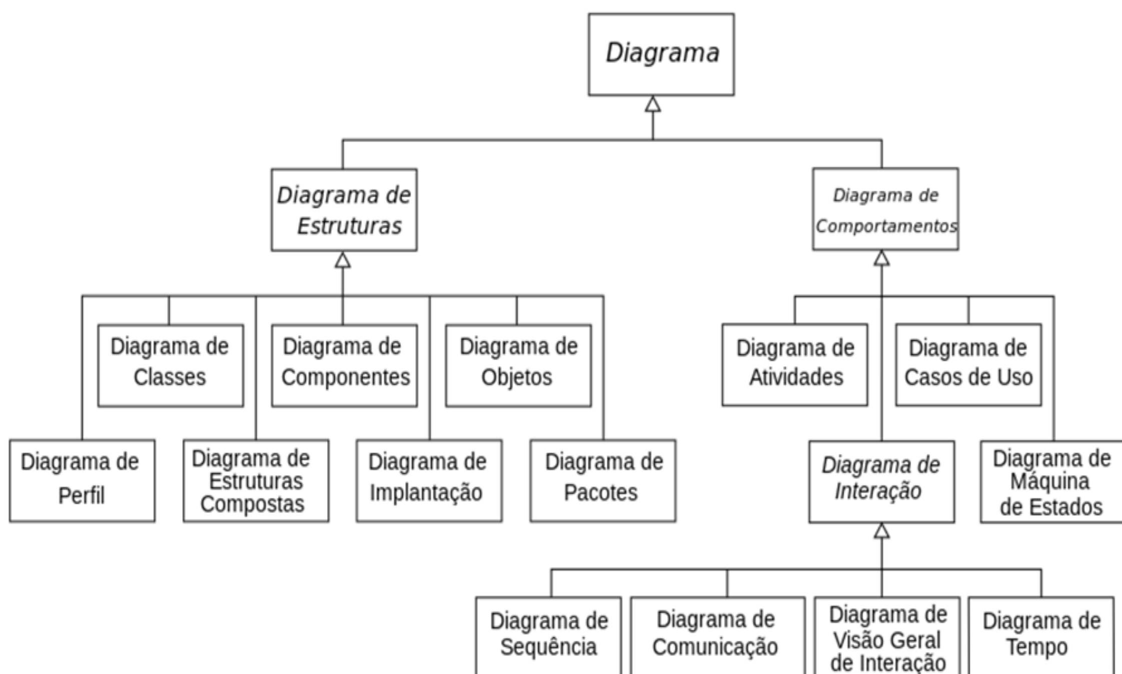

Figura 6: Diagramas estruturais e comportamentais da UML [41].

De acordo com Sommerville [23], a essência de um sistema pode representado por cinco diagramas da UML: o diagrama de casos de uso, o diagrama de atividades, diagrama de classes, o diagrama de sequência e o diagrama de estados.

- Diagrama de casos de uso: mostra as interações entre um sistema e o ambiente. Auxilia na identificação e compreensão dos requisitos do sistema, ajudando a especificar, visualizar e documentar as funções e serviços do sistema desejados pelo usuário [40].
- Diagrama de atividades: mostra as atividades envolvidas em um processo ou no processamento de dados [23]. O diagrama de atividades enfatiza a sequência e condições para coordenar comportamentos de baixo nível. É provavelmente um dos mais detalhistas [40].
- Diagrama de classes: mostra as classes de objeto e as associações entre elas [23]. Apresenta os atributos e métodos do objeto instanciado de uma classe. Os métodos podem ser demonstrados a partir de diagramas de interação [40].
- Diagrama de sequência: mostra as interações entre os atores e o sistema e seu ambiente [23]. Busca determinar a sequência de eventos que ocorrem em um determinado processo, identificando quais mensagens devem ser disparadas entre os elementos envolvidos e em que ordem [40].
- Diagrama de estados: mostra como o sistema reage aos eventos internos e externos [23]. O diagrama demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado [40].

Este trabalho objetiva estabelecer um processo de desenvolvimento apoiado nos aspectos e características do MDA para a modelagem de scripts e sistemas de automação de testes funcionais para sistemas embarcados, utilizando para isso a UML, mais especificamente os diagramas de Classes, Objetos e Sequência.

O MDA é uma arquitetura de desenvolvimento orientada a modelos proposta e mantida pela OMG e que baseia-se fundamentalmente no padrão de metamodelagem MOF. Sendo a UML a linguagem de modelagem mais conhecida derivada do MOF, optou-se por utilizar modelos nativos dessa linguagem. Uma vez que os três diagramas adotados são baseados em um mesmo metamodelo, a interação e intercâmbio de informações entre os mesmos é facilitada devido ao núcleo comum existente entre eles.

3.8.1 Diagrama de Classes

O diagrama de Classes integra na tipologia dos diagramas estruturais, e representa segundo Guedes [40], um dos diagramas mais importantes da UML. Exerce função de apoio à maioria dos demais diagramas, uma vez que define a estrutura das classes que compõem o sistema assim como seus atributos, métodos e relacionamentos. Os principais elementos presentes neste tipo de diagrama, conforme definidos pela OMG [41]:

Abstraction: A Classe *Abstraction* representa o conceito de abstração da UML, sendo definida como um relacionamento entre dois elementos que possuem um mesmo conceito em níveis diferentes de abstração.

Association: Uma *Association* especifica uma relação semântica que pode ocorrer entre instâncias tipadas. Ela possui ao menos duas extremidades representadas por propriedades, cada uma das quais estando ligada ao tipo da extremidade conhecidas como *AssociationEnd*.

AssociationClass: Uma *AssociationClass* pode ser vista de duas formas distintas, como uma classe que possui características de uma *Association*, ou uma *Association* que possui características de uma classe, como *Features*, nome, etc. Ou seja, uma *AssociationClass* é ao mesmo tempo uma *Association* e uma *Class*.

BehavioralFeature: Um *BehavioralFeature* é uma característica de um classificador que especifica um aspecto do comportamento de suas instancias. Ou seja, as instâncias de um dado *Classifier* respondem a uma dada requisição através da invocação de um comportamento esperado.

Classifier: Um *Classifier* define uma classificação de instâncias de acordo com as características que as mesmas detêm.

Class: Uma *Class* descreve um conjunto de objetos que compartilham características, restrições e aspectos semânticos semelhantes. Uma *Class* é uma especialização de um *Classifier*, sendo que seus *Features* são atributos e métodos.

Constraint: Uma *Constraint* é uma condição ou restrição imposta a um elemento. Geralmente é expressa sob a forma de linguagem natural, mas em alguns casos pode aparecer na forma de alguma linguagem de máquina.

DataType: Um *DataType* é um tipo de dado cujas instâncias somente podem ser identificadas pelo seu valor. O uso mais comum para um *DataType* é para representar tipos primitivos de uma determinada linguagem de programação como *integer* ou *string*.

Dependency: Uma *Dependency* é um tipo de relacionamento entre elementos de um modelo que especifica a condição de que um dado elemento ou conjunto de elementos são dependentes de outros elementos do modelo para serem totalmente especificados ou implementados.

Element: A classe *Element* é a super-classe de todas as metaclasses contidas na biblioteca de infraestrutura da UML. Ou seja, um *Element* é um componente de um modelo, que pode ainda conter outros elementos.

Feature: Uma *Feature* declara uma característica comportamental ou estrutural de uma instancia de um *Classifier*.

Generalization: Uma *Generalization* é um relacionamento sistemático existente entre dois *Classifiers*, um mais geral e outro mais específico. Cada instância do *Classifier* mais específico é também uma instância do *Classifier* mais geral, ou seja, o *Classifier* mais específico herda as características do *Classifier* mais Geral.

NamedElement: Um *NamedElement* representa um elemento que pode ter um nome. O nome é usado para a identificação do *NamedElement* dentro do *NameSpace* no qual ele é definido.

Namespace: Um *Namespace* é um *NamedElement* que pode possuir outros *NamedElements*. Cada *NamedElement* pode estar contido em no máximo um *Namespace*. Um *Namespace* fornece um meio para identificar *NamedElements* pelo nome.

Operation: Um *Operation* é uma característica comportamental de um *Classifier* que especifica o nome, tipo, parâmetros e restrições necessárias para invocar um comportamento associado à operação.

Parameter: Um *Parameter* é uma especificação de um argumento usado para passar informações para dentro ou fora de uma invocação de uma característica comportamental.

A Figura 7 apresenta as representações gráficas, quando existentes dos elementos de um diagrama de classes.

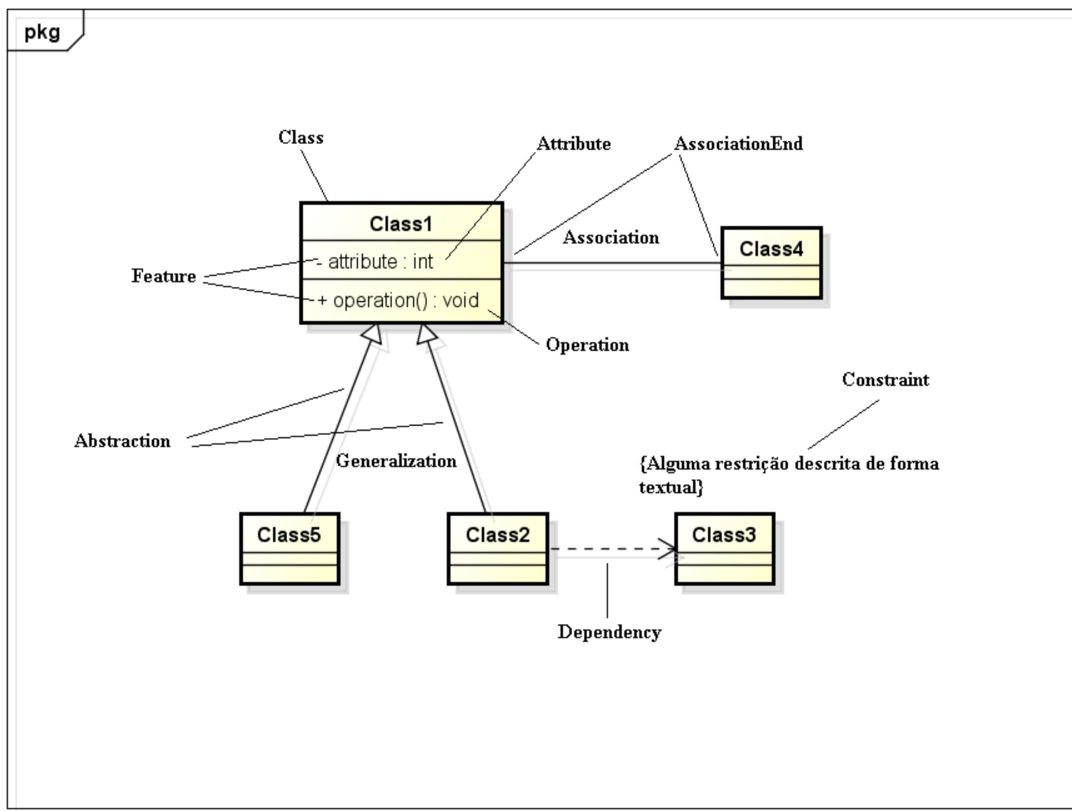

Figura 7: Principais elementos construtivos do diagrama de classes da UML.

3.8.2 Diagrama de Sequência

O Diagrama de Sequência apresenta através de seus elementos, as interações entre atores, objetos e suas classes por meio da troca de mensagens e operações entre eles [42]. A seguir são definidos, de acordo com [41] e [43], os elementos construtivos presentes nesse tipo de diagrama:

• ***interaction***: é uma unidade de comportamento que foca na troca de informações entre os elementos conectados;

• ***interactionFragment***: é uma noção abstrata de uma unidade de interação mais genérica. Ou seja, uma *interactionFragment* pode ser entendida como uma parte ou peça de uma *interaction*;

• ***interactionOperator***: é um operador indicado na criação de um *combinedFragment* que atribui características específicas a uma dada *interactionFragment*. Um *interactionOperator* pode ser englobado nos seguintes tipos: *alt, opt, par, loop, critical, neg, assert, strict, seq, ignore* ou *consider;*

•***interactionOperand***: representa um operador de um *combinedFragment*;

•***combinedFragment***: define uma expressão do *interactionFragments*. Um *combinedFragment* é definido por *interactionOperator* e *interactionOperands* correspondentes;

• ***executionOccurrenceSpecification***: representa o momento no qual ações ou comportamentos iniciam ou terminam;

• ***lifeline***: representa um participante individual na *interaction* e seu tempo ou linha de vida;

• ***message***: define uma comunicação particular entre *lifelines* e/ou uma *interaction*;

• ***executionSpecification***: é uma especificação da execução de uma unidade de comportamento ou ação interna da *lifeline*. A duração de uma *executionSpecification* é representada por duas *executionOccurrenceSpecifications*, uma inicial e outra final;

• ***occurrenceSpecification***: é uma unidade de semântica básica de interações. *OccurrenceSpecification* são apenas pontos sintáticos nas extremidades das mensagens ou no início/fim de uma especificação de execução.

A Figura 8 apresenta a representação gráfica de alguns dos elementos citados.

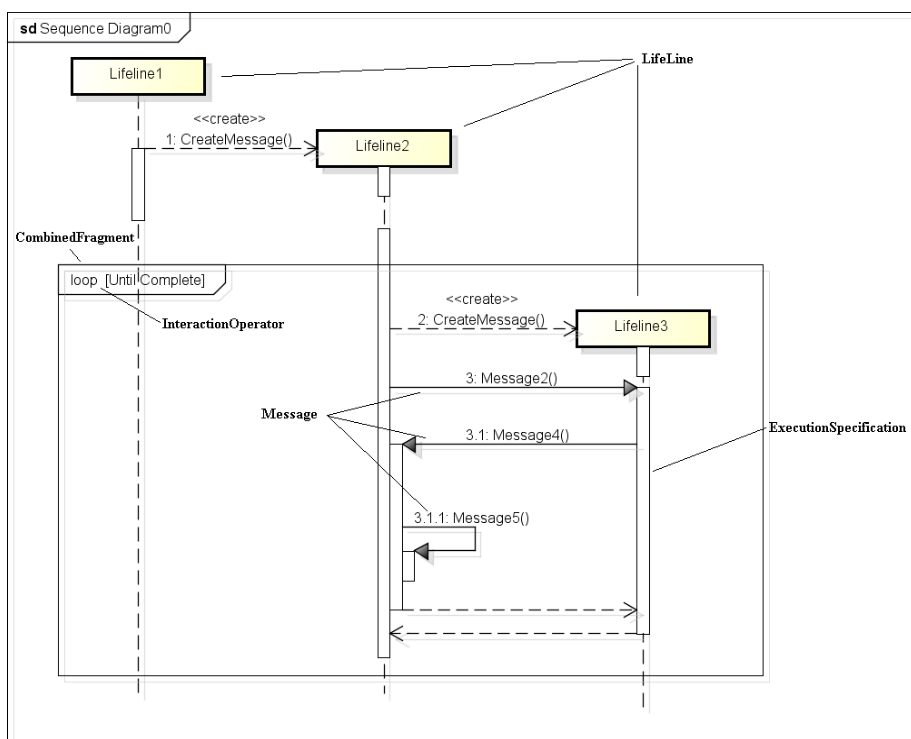

Figura 8: Principais elementos construtivos do diagrama de sequência da UML.

3.2 RELAÇÃO MOF x UML

O MOF define uma linguagem abstrata para definir linguagens de modelagem ou ainda metamodelos. Como seus elementos construtivos são, em sua maioria, semelhantes aos construtores da UML, pode-se utilizá-la para representá-lo graficamente. A OMG adota as representações gráficas assim como diversos pacotes da UML, como *InfraStructure Library*, *Core* e *Kernel*, para definir o próprio MOF, criando assim uma relação recursiva, em que o MOF define a metalinguagem do modelo UML, que por sua vez é utilizado para representação da especificação do MOF. No entanto, é importante salientar que, semanticamente o modelo MOF é totalmente independente de quaisquer outros modelos definidos a partir do mesmo, inclusive da UML.

Enquanto a UML é uma linguagem de modelagem universal adotada pela OMG como, o padrão para modelagem de sistemas discretos, o MOF é o padrão adotado para definição de definir linguagens de modelagem. Nessa seção, apesar da tratativa sobre a relação existente entre o MOF e a UML, ressalta-se que o MOF é também base para a definição de outros metamodelos empregados pela OMG, como o CWM e o XMI [44].

As diferenças estruturais entre os elementos construtores da UML e do MOF, segundo [38] são as seguintes:

1. O MOF suporta apenas associações binárias, ao passo que a UML suporta associações n-árias. Essa diferença se deve basicamente ao fato de que em um nível mais abstrato de metamodelagem, raramente se usam associações entre mais de dois elementos.
2. O MOF não promove suporte a classes associativas, i.e., as associações previstas pelo MOF não possuem características relacionadas a Classes como *features* ou nome, aspecto existente na UML.
3. O MOF suporta o conceito de referência que permite a navegação entre *Classifier* de forma direta. A UML por sua vez faz uso do nome do *AssociationEnd* alvo como um "pseudo-atributo" para esse fim, que resulta na navegação por meio de uma *Association*.
4. Alguns conceitos como generalização e dependência são especificados na forma de classes na UML e em forma de associações no MOF.
5. A UML define de forma clara as semelhanças entre os conceitos de classe, interface e tipo.

Apesar dessas divergências, na atualidade, a UML é a linguagem de modelagem adotada para representar graficamente metamodelos MOF.

3.3 Ferramentas de MDA

O desenvolvimento orientado por modelos baseia-se na premissa de que os modelos possuem um papel de vital importância no processo de desenvolvimento. Apesar de a UML ser independente de ferramenta, as mesmas certamente desempenham um papel importante para a redução dos erros e a diminuição de custos no processo de desenvolvimento [45]. A evolução de muitas ferramentas UML foi resultado do esforço por atender às necessidades da abordagem de desenvolvimento orientado a modelos e MDA.

O trabalho de Caliari [46] realiza um estudo sobre os conceitos do MDA e faz um paralelo entre três ferramentas, OptimaJ, AndroMDA e ArcStyler, focando a forma como as mesmas implementam, fazem uso desses conceitos e ainda destaca a abordagem utilizada por cada uma dessas ferramentas para as transformações entre modelos e também geração de código. Por fim, o autor realiza um estudo de caso com o intuito de avaliar qualitativamente como os mapeamentos definidos nas ferramentas são utilizados nas transformações. Segundo o autor, cada uma das ferramentas apresenta vários pontos positivos no que tange às implementações de conceitos MDA, no entanto nenhuma delas implementa a totalidade de conceitos descritos no padrão estipulado pela OMG. Em parte, esse cenário é atribuído ao fato de que, embora o MDA seja um paradigma de desenvolvimento, o mesmo não especifica um padrão para o uso de seus conceitos, possibilitando assim que o desenvolvimento de software se torne dependente de uma ferramenta específica, uma vez que não existe uma formalização sobre como devem ser aplicados os conceitos do MDA.

Calic, Dascalu e Egbert [47] propõem as características desejadas para uma ferramenta MDA, como:

1. Prover suporte à UML e suas extensões para modelagem de PIM;
2. Gerar automaticamente modelos PSM a partir de modelos PIM;
3. Prover meios para realização de modificações em modelos PSM para atender a necessidades específicas de um projeto ou usuário;
4. Suportar transformações de modelo para outros modelos assim como de modelos para códigos;
5. Suportar meios de depuração para transformações entre modelos;
6. Gerar automaticamente a documentação de um projeto;

7. Possibilitar a exportação de modelos no formato XMI, que facilite a interoperabilidade entre ferramentas MDA;
8. Suportar processos de engenharia reversa para geração de modelos a partir de códigos de sistemas legados;
9. Prover formas de integração com IDE's (*Integrated Development Environment)* específicas de linguagens como Eclipse e Visual Studio;
10. Prover suporte para trabalho coletivo, possibilitando que vários colaboradores atuem de forma simultânea em um projeto;
11. Possuir uma documentação completa e guia de usuário com exemplos e explicações sobre as funcionalidades da ferramenta.

3.4 Trabalhos Relacionados

Nesta seção são discutidos os trabalhos relacionados ao tema desse estudo, focado nos avanços e trabalhos já realizados junto à temática de automação de testes de software e MDA. Para tanto foi realizado um levantamento bibliográfico de artigos científicos pertinentes ao tema, assim como um mapeamento sistemático da literatura. Os resultados dessas atividades de pesquisa são apresentados em duas subseções, uma voltada ao tema de automação de testes especificamente aplicados em sistemas embarcados, e outra subseção dedicada à explanação e resultados do mapeamento sistemático da literatura.

3.4.1 Automação de testes em sistemas embarcados

Bringmann e Kramer [28] descrevem a utilização da ferramenta de testes TPT (*Time Partition Test*), desenvolvida para modelagem e execução automática de testes. Os autores utilizam como estudo de caso um sistema fictício de controle de acendimento automático de farol de acordo com o índice de luminosidade do ambiente. A linguagem de modelagem adotada pelos autores para a TPT consiste em uma linguagem gráfica baseada nos conceitos de máquina de estados, sendo que aos estados do modelo, são associadas equações matemáticas que descrevam a forma como os sinais de entrada são estimulados, e às transições, são associadas condições temporais, que executam a mudança entre dois estados distintos. Almeida e Travassos

Huang [48] desenvolveu em conjunto com outros autores, um trabalho utilizando técnicas de Processamento Digital de Imagens - PDI para execução de testes automatizados em painéis automotivos. A execução automatizada dos testes é (HIL) realizada com base em scripts de automação gerados na linguagem de programação Python a partir de uma descrição textual do caso de teste feita em planilhas de Excel e salvas em formato .xls.

Yin Yongfeng, Liu Bin e Zheng Bentao [49] descrevem uma linguagem de scripts baseada nos conceitos de orientação a objetos, desenvolvida pelos autores como forma de reduzir a complexidade na criação de scripts de automação de teste para sistemas embarcados, que geralmente seguem um padrão estrutural e sequencial de programação. Além disso, os autores apontam a utilidade dessa mesma linguagem de scripts como meio de modelagem de simulações de ambientes e outros dispositivos que possam interagir com o SUT, emulando assim os estímulos captados pelo SUT provenientes do ambiente para o qual o mesmo foi projetado para operar.

Obele [50] sugere uma forma alternativa para a automação de testes, propondo uma arquitetura de software que disponibiliza interfaces usadas para que ferramentas de automação de teste possam se comunicar com o SUT, simulando as ações finais do usuário como comandos de voz e pressionar de botões e capturando os comportamentos em resposta do SUT. No entanto, os próprios autores apontam limitações dessa proposta no que tange ao aumento da complexidade do software. Apesar disso, foi apresentado um estudo de caso no qual esse método foi aplicado no desenvolvimento de um software de controle de um dispositivo multimídia para automóveis, sendo posteriormente utilizada uma ferramenta para execução de testes automatizados no SUT.

Moon et al. [51] utilizam Matlab e Simulink para criar ambientes simulados de teste apoiados pela arquitetura proposta pela norma AUTOSAR. Apresentam um estudo de caso onde realizam a automação de testes relativos a um sistema de controle de temperatura de interior em automóveis composto por ar condicionado e aquecedor.

Percebe-se que os trabalhos apresentados nesta seção focam o processo de automação de casos de teste criados manualmente pela equipe de testes, utilizando para isso técnicas diversas para criação de scripts de automação ou modelos comportamentais para esse fim. No entanto, um fator importante para a automação de testes em sistemas embarcados e que foi abordado por alguns desses trabalhos está relacionado à simulação do ambiente no qual o SUT deverá atuar, assim como os estímulos fornecidos ao mesmo por esse ambiente simulado. Os estudos de Bringman e Kramer, Huang, Yin Yongfeng, Liu Bin e Zheng Bentao e Moon abordam esse tema, propondo formas de implementar as simulações, sem entretanto

realizar a devida modelagem desses ambientes. Além disso, todos os trabalhos apresentam soluções voltadas a um ramo específico no qual os sistemas embarcados se fazem presentes, principalmente o setor automotivo.

3.4.2 Mapeamento Sistemático

Nesta seção será abordado o mapeamento sistemático de literatura realizado. A metodologia adotada para este mapeamento foi baseada nas diretrizes propostas por Petersen et. al. [52].

Dessa forma, o mapeamento foi realizado contemplando as bibliotecas digitais da IEEE e ACM por meio de mecanismos de pesquisa automatizados e uso de uma *string* de palavras chaves elaborada a partir das principais palavras e objetivos deste trabalho. Deste modo, foi gerada a seguinte *string,* a qual foi utilizada nas buscas:

"(((model driven architecture) OR MDA) AND test automation) OR ((model driven architecture OR MDA) AND CIM transformation) OR (test automation tools AND UML) OR (test automation tools development)"

Para filtrar o resultado obtido, foram definidos critérios para inclusão e exclusão dos trabalhos obtidos:

- Exclusão: Trabalhos repetidos; Trabalhos publicados anteriormente ao ano de 2005.
- Inclusão: Trabalhos diretamente relacionados com a Automação de testes e/ou MDA; Trabalhos que tenham relação com os objetivos traçados neste estudo;

Após a realização das buscas nas referidas bibliotecas, foram encontrados 63 trabalhos, dos quais sete foram listados nos resultados da biblioteca do IEEE e 56 na biblioteca da ACM. Adiante a verificação da primeira triagem, 17 trabalhos foram selecionados a partir dos critérios de inclusão e exclusão para uma análise mais detalhada, sendo os mesmos classificados em 6 categorias de acordo com o tema da pesquisa. A tabela 2 apresenta uma essa categorização juntamente com o número de trabalhos associados a cada categoria.

Tabela 2: Resultado e classificação dos trabalhos analisados.

<table>
<tr><th>Tema</th><th colspan="3">Resultado da string da busca</th><th>Após triagem (Inclusão / Exclusão)</th><th colspan="2">Categorias utilizadas para Classificação</th></tr>
<tr><td rowspan="6">Automação de testes e/ou MDA</td><td rowspan="3">IEEE</td><td rowspan="3">7</td><td rowspan="6">63</td><td rowspan="6">17</td><td>Aplicação de MDA em algum domínio específico</td><td>4</td></tr>
<tr><td>Geração automática de caso de testes a partir de MDA</td><td>3</td></tr>
<tr><td>Utilização de MDA para criação de testes</td><td>1</td></tr>
<tr><td rowspan="3">ACM</td><td rowspan="3">56</td><td>Ferramentas para transformação</td><td>2</td></tr>
<tr><td>Surveys ou revisões sistemáticas</td><td>4</td></tr>
<tr><td>Outros</td><td>3</td></tr>
</table>

Dentre os trabalhos analisados, 4 apresentam estudos ou pesquisas a respeito da aplicação do MDA em um domínio específico de problema como no desenvolvimento de softwares militares e para o setor automotivo; 3 trabalhos abordam o tema de geração de casos de teste de forma automática a partir de artefatos gerados pela aplicação do MDA; 2 trabalhos apresentam os resultados de pesquisas voltadas à criação de ferramentas de suporte ao MDA, principalmente no que diz respeito à criação e realização de transformações automáticas entre modelos; 4 trabalhos apresentam resultados de *surveys* e revisões sistemáticas a respeito do MDA; 1 trabalho aborda a utilização do MDA para o desenvolvimento de casos de teste em proposta similar a deste estudo e outros 3 trabalhos abordam outras iniciativas relacionadas a teste de software em conjunto com o MDA. A seguir é feita uma breve explanação sobre o conteúdo de cada um desses estudos organizados pela temática relacionada a este trabalho.

Aplicação de MDA em algum domínio específico:

Ferreira [53] aborda em seu trabalho um estudo voltado para a indústria automotiva e apresenta uma variante do MDA intitulada MDAI, que objetiva transformação entre modelos PIM para PSM, sem abordar os modelos CIM. Para isso são utilizados os padrões XMI, XML, VDML e UML. Por fim, também é realizada a transformação de PSM para código, no caso código em CatiaV usado para verificação de ergonomia em componentes automotivos.

O trabalho realizado por Gailliard et al. [54] apresenta o uso do MDA e suas transformações de PIM para PSM no contexto de desenvolvimento de aplicações militares, mais especificamente para a criação de um software para controle de dispositivos geradores de ondas de radiofrequência com diferentes formas de onda. Para isto são utilizados os diagramas UML de classes e sequência para a modelagem do software. Além disso, é apresentada também uma ferramenta de simulação de modelos com o intuito de simular e validar os modelos PIM e PSM após as transformações. Este trabalho assim como o anterior não aborda os modelos CIM.

Outro trabalho aborda a utilização de conceitos do MDA como CIM, PIM e PSM, que empregam, no entanto uma base diferente da proposta pela OMG e baseados no MOF. Neste estudo Sindico et al. [55] usa o metamodelo *Ecore* como base para a definição de um processo de desenvolvimento completo. De acordo com os autores, como resultado 90% de todo o código do sistema é gerado automaticamente sendo que o código comportamental é gerado por meio da ferramenta Symulink e o código estrutural é gerado a partir de modelos SysML e Ecore.

Guttman e Parodi [56] apresentam em seu livro casos onde o MDA foi aplicado, com os motivos e como o mesmo foi aplicado. Os autores também relatam a dificuldade de transformação de CIM para PIM e que o próprio MDA não apresenta muita informação de como proceder neste processo de transformação, mas reforçam a importância do CIM para o entendimento do domínio da aplicação e para descrever as funcionalidades do sistema.

Geração automática de casos de teste a partir do MDA:

Neste contexto de geração automática de casos de teste a partir de MDA o trabalho de Javed et. Al [57] propõe a utilização de transformações MDA em níveis de modelo para modelo e modelo para código no contexto de Verificação e Validação, que geram casos de teste a partir do PIM do sistema. As transformações de modelo para modelo são utilizadas para transformar modelos de diagrama de sequência do software em modelos específicos para execução de testes por intermédio da aplicação de tecnologia XUnit de testes unitários. Por fim, a esse modelo é aplicada outra transformação para torna-los de modelos PIM para PSM, visando plataformas específicas do XUnit como o JUnit.

O estudo de Carrozza et al. [58] propõe uma integração entre MDA e MDT (*Model Driven Testing*), por meio do emprego de transformações entre modelos PIM para PIT (*Platform Independent Test*). A partir dessa abordagem, os autores propõem que os casos de

teste sejam gerados automaticamente por transformações MDA a partir de diagramas comportamentais do sistema em UML. Os casos de teste gerados a partir dessa transformação são representados na forma de diagramas de sequência, que por fim geram um *script* para execução do teste em uma linguagem chamada TTCN-3. Além disso, este trabalho aborda a necessidade da criação manual de um adaptador para execução do script de teste gerado, para transformar os comandos do script em mensagens enviadas ao SUT.

Alves et al. [59] apresenta um método de geração automática de casos de teste denominado *Built in contract testing* (BIT), assim como uma extensão da UML criada pelos autores de mesmo nome. Nesta abordagem, componentes de software são modelados via modelos UML nos quais são inseridas anotações desse *profile* que são utilizadas para gerar regras de transformações de modelos PIM para modelos de teste independentes de plataforma. É descrita ainda a criação de uma ferramenta chamada MoBIT que faz as transformações de PIM para PSM e também para modelos de teste a partir de diagramas de estado e sequência da UML.

Utilização de MDA para criação de testes:

No trabalho proposto por Wang et al. [60] são criados scripts para serem executados em uma *engine* de teste. A transformação de CIM para PIM é feita de forma manual, usando um gerador de sequência de teste como ferramenta para auxiliar o processo, mas não especifica detalhes sobre o processo de transformação.

Ferramentas para transformação:

No trabalho de Edwards e Gruner [61] é descrito a criação de uma ferramenta para criação de transformação semiautomática de PIM para PSM de modelos criados a partir do método "*Use Case, Responsibility Driven Analysis and Design*" (URDAD), desenvolvido pelos autores. A ferramenta desenvolvida aborda especificamente transformações e mapeamentos do URDAD PIM para o PSM EJB (*Enterprise Java Beans*) que gera o código a partir deste último modelo.

Bollati et al. [62] apresenta um *framework* chamado MeTaGem que consiste de uma proposta metodológica para criação de transformações para modelos para auxiliar no processo de desenvolvimento, utilizando os conceitos de MDD. Ou seja, os autores propõem a utilização de conceitos e técnicas do MDD para a criação de transformações de modelos em software.

Surveys e revisões sistemáticas:

O trabalho de Mohagheghi et al. [63] reúne experiências de grandes empresas europeias no desenvolvimento de técnicas e ferramentas para aplicação do MDE

(*Model-Driven Engineering*) no desenvolvimento de software grande e complexo. No ponto de vista das empresas a MDE auxilia na comunicação entre os membros da equipe de desenvolvimento, além de permitir reutilização de soluções de outros projetos ou produtos. Porém apresentam que em alguns casos a geração de soluções reutilizáveis necessitou de um esforço maior que impactou negativamente no esforço das ferramentas. Por fim, o relato das empresas acaba por identificar vários benefícios da utilização de MDE para desenvolvimento de software de grande porte e complexo.

No *survey* de Valderas e Pelechano [64], é apresentada o estado da arte de técnicas de especificação de requisitos usando (MDD) para aplicações *web*. Neste estudo são classificados alguns métodos MDD por ordem de facilidade. Também é feita uma comparação entre outros trabalhos relacionados ao tema MDD.

Loniewsky et al. [65] apresenta uma revisão sistemática do uso de processos MDD no desenvolvimento de software e seus atuais níveis de automatização. Neste trabalho foram analisados 65 artigos do período de 2001 a 2010 que estão relacionados ao tema abordado. Os resultados do trabalho apresentam que apesar dos modelos PIM e PSM serem bastante utilizados modelos de definição de requisitos são parcialmente usados ou substituídos por linguagem natural.

Hebig e Bendraou [66] apresentam outra revisão na literatura que trazem trabalhos que utilizaram técnicas de MDE e outros padrões como SCRUM e Modelo V, e mostram casos de abordagens de sucesso com uso de reutilização de processos sem nenhum tipo de adaptação que outros que utilizam uma combinação de técnicas MDE e como elas podem trazer benefícios em termos de economia de tempo e custos.

Outros:

Almeida e Oliveira [67] apresentam um modelo de processo de desenvolvimento de software chamado de Qualitas no qual os autores buscam integrar características de modelo espiral de desenvolvimento com os conceitos de MDA e MDT. O processo proposto utiliza os modelos CIM, PIM e PSM, no entanto não são abordadas questões relacionadas às transformações automáticas entre esses modelos.

O trabalho de Diaz et al. [68] apresenta um *framework* denominado TALISMAN MDE, que busca fazer uma integração entre MDA e o conceito de fábricas de software, que visa maximizar a produtividade proporcionada pelas fábricas de software com a interoperabilidade, portabilidade e reusabilidade da proposta MDA. O *framework* consiste de um processo de criação de software usando os conceitos de CIM, PIM e PSM e

transformações MDA associadas ao foco das fábricas de software em desenvolver famílias de software e linhas de produtos baseadas em um domínio específico de aplicação.

Normalmente, sistemas de automação de teste são criados com o auxílio de sistema ou ferramentas desenvolvidas para este fim, e que suportam uma ou mais linguagem para a criação de scripts de teste que são responsáveis por executar a sequência de ações de um caso de testes. Fazendo uso dessas ferramentas, testadores tem a responsabilidade de traduzir casos de teste escritos de forma textual em scripts de teste para uma dada linguagem de teste.

Um ponto a se observar neste levantamento é a dificuldade na realização da transformação automática de CIM para PIM, são poucos os trabalhos que abordam este tipo de transformação, a maioria parte da transformação PIM-PSM-Código e os que trabalham com transformação a partir do CIM a fazem de forma manual. Isto se deve a não existência de artefatos e/ou regras específicas que definam quais elementos devem ser utilizados no CIM e consequentemente não há uma normatização para o mapeamento de CIM-PIM, isto pode ser observado em diversos trabalhos [47], [54], [56], [59].

Por fim, com o intuito de sumarizar os trabalhos analisados neste mapeamento sistemático, foi gerado ainda um gráfico de bolhas, apresentado na Figura 9, onde é possível observar a relação existente entre a quantidade de trabalhos sobre cada tema ao longo de uma linha temporal. É possível perceber que apesar de possuir uma relativa constância em termos de estudos, o tema ainda foi pouco explorado dentro da proposta específica deste trabalho que é a de aplicar os conceitos e diretrizes do MDA para o contexto de desenvolvimento de ferramentas de automação de testes, visto que essa temática apresenta somente um trabalho publicado no período escolhido para o mapeamento.

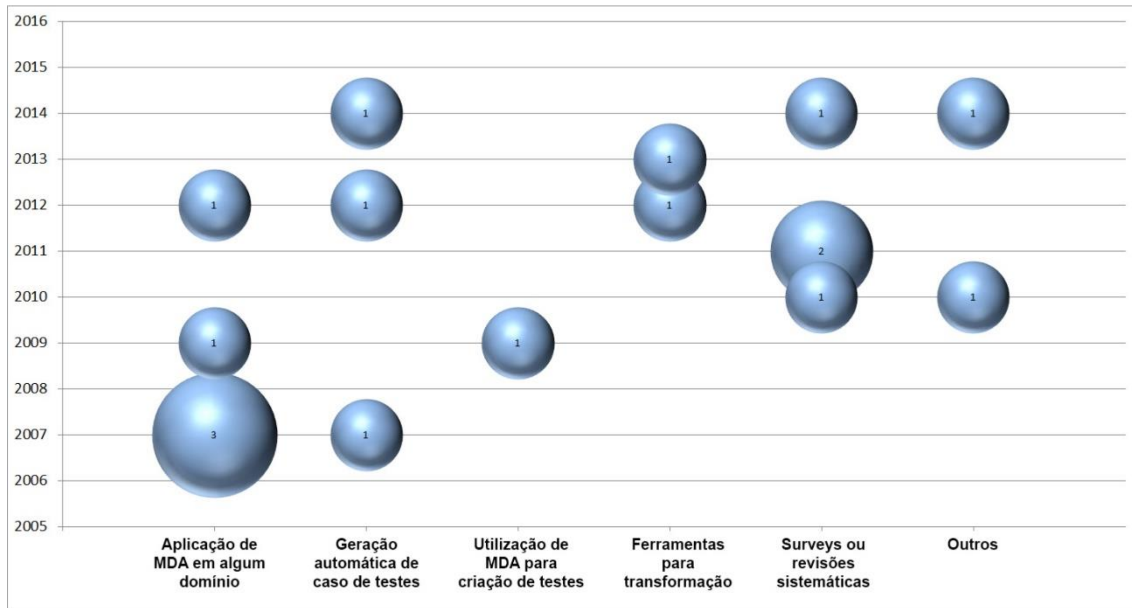

Figura 9: Relação entre temas quantidade de artigos por ano de publicação.

4 PROPOSTA DE UM MDA PARA DESENVOLVIMENTO DE SISTEMAS DE AUTOMAÇÃO DE TESTES

A proposta deste trabalho é criar um processo de desenvolvimento de sistemas de automação de testes funcionais para sistemas embarcados, estabelecendo meios para transformação dos documentos de teste em modelos de casos de teste. Para isso foram estabelecidos alguns formatos de documentos como a definição de uma sintaxe para escrita dos casos de teste, um formulário de dados que sumarize as informações necessárias para a caracterização dos teste, além da disponibilização de uma estrutura de *Framework* básica para a execução automatizada de testes de software embarcado pautados na aplicação de conceitos e técnicas do MDA. Além disso, o processo criado propõe ainda que seja realizada a inserção de elementos que modelem os componentes/entidades responsáveis por capturar o estado do dispositivo em teste automaticamente. A figura 10 mostra uma versão simplificada do processo proposto, dando ênfase aos tipos de modelos, CIM, PIM e PSM, assim como descrevendo suscintamente as atividades relacionadas a cada um deles. Além disso, a Figura 10 evidencia quais etapas do processo serão realizadas de forma automática, manual ou ainda, etapas passíveis de automação mediante a criação de ferramentas para esse fim.

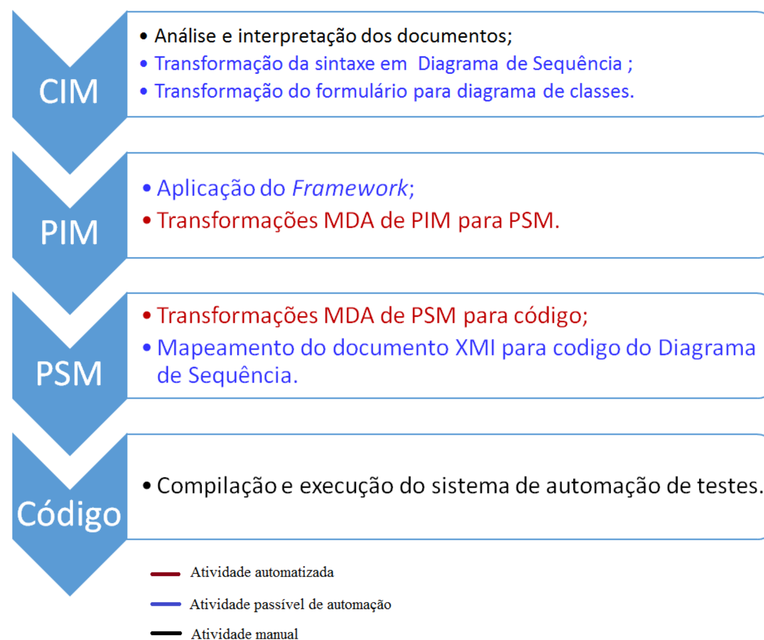

Figura 10: Processo proposto para desevolvimento de sistemas de automação de testes funcionais.

A aplicação das técnicas de MDA busca oferecer nesta proposta:

- Aplicação do paradigma de desenvolvimento orientado a modelos ao desenvolvimento de sistemas de automação de teste;
- Meios para estruturar o entendimento dos casos testes a serem realizados a partir de uma melhor organização e normatização dos documentos em CIM, possibilitando transformações MDA de CIM para PIM;
- Padronizar o procedimento adotado para o desenvolvimento de ferramentas de automação de testes;
- Agilizar o desenvolvimento de novas ferramentas de teste, uma vez que muitos testes são podem possuir características semelhantes;
- Um meio para gerar reuso de componentes de software, visto que testes de software estão sujeitos a ser realizados diversas vezes em função de atualizações do software testado;
- Facilitar a alteração ou atualização da ferramenta de teste frente a alterações ou atualizações do SUT.

O escopo deste trabalho aborda o processo de automação dos testes, não abrangendo a criação dos casos de teste. Dessa forma, a proposta deste estudo inicia-se a partir da hipótese de que a equipe de testes já possui previamente um documento com as informações necessárias à realização dos testes, os quais normalmente são realizados de forma manual.

De acordo com a argumentação apresentada por Kassurinen [7], os documentos de teste sofrem com a falta de padronização em sua criação, uma vez que é comum encontrar cenários onde cada empresa possui um formato próprio de documentação e escrita de testes, não existindo um padrão efetivo e amplamente difundido. Diante disso, este trabalho desenvolveu uma forma padronizada de interpretar e organizar os documentos de teste com o intuito de melhor adequá-los para a aplicação de transformações automáticas requeridas pelo MDA.

Além disso, Kriouile, Addamssiri, e Gadi [69] salientam que dentro do processo de desenvolvimento proposto nas diretrizes do MDA, as transformações automáticas entre CIM e PIM são pouco abordadas, dada a grande distância em termos conceituais existente entre tais modelos.

Com o intuito de reduzir a lacuna entre os CIM e os PIM, neste trabalho foi desenvolvida uma sintaxe que possibilite uma melhor escrita e organização dos casos de teste, e que ainda, viabilize a padronização dos documentos de teste, tornando factível a aplicação de transformações automáticas de CIM para PIM dentro do domínio de aplicações voltadas para a automação de testes funcionais em sistemas embarcados.

Desta forma, visto que o MDA adota principalmente a UML como linguagem de modelagem, neste trabalho foram empregados como modelos alvo das transformações de CIM para PIM os diagramas UML de Classes para modelagem estrutural das entidades que compõem o sistema de testes e o diagrama de Sequência para a modelagem dos *scripts* de automação. Esses elementos são criados objetivando-se posteriores transformações automáticas da MDA para a geração do código do sistema.

Assim, foi criado um processo de desenvolvimento para a criação de ferramentas de automação de testes pautado nas diretrizes da MDA e apresentado na Figura 9.

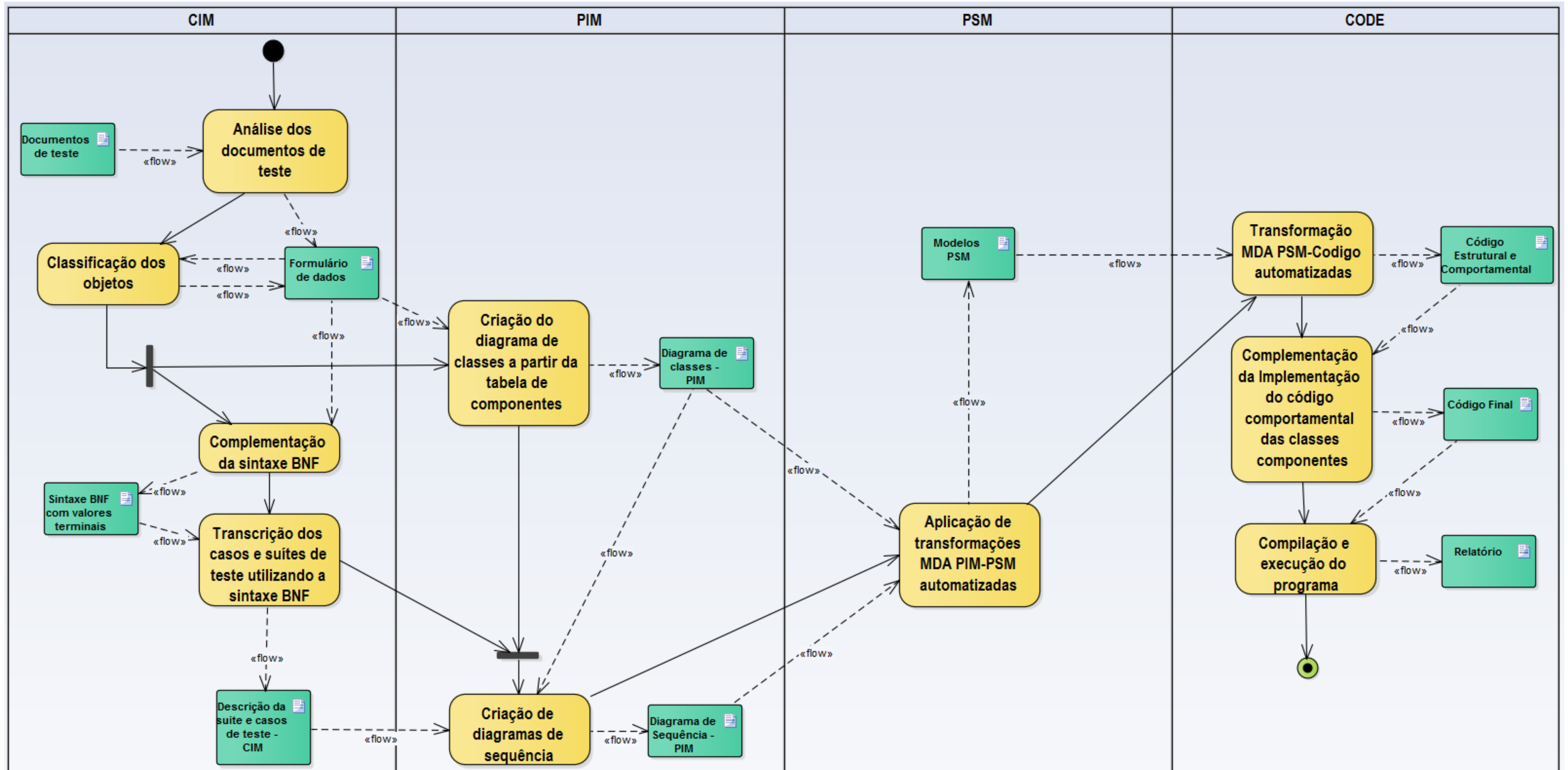

Figura 11: Processo de desenvolvimento proposto.

O diagrama de atividades da figura 11 foi dividido em raias com o intuito de melhor organizar os artefatos e atividades do processo, possibilitando ainda uma melhor correlação de cada elemento com os respectivos modelos propostos pelo MDA. Assim, o processo inicia-se com as atividades relacionadas com as análises e transformações de documentos de teste, nesse caso tratados como documentos CIM. Sequencialmente, as fases de transformações do MDA de modelos CIM para PIM e de PIM para PSM são abordadas, finalizando-se o processo com as atividades de geração de código e execução do sistemas modelado. Cada uma das atividades listadas no processo da Figura 11 serão abordadas e descritas em maiores detalhes nas seções subsequentes.

3.1 Modelo independente de Computação – CIM

Os modelos do CIM são relacionados ao domínio da aplicação, A presente proposta visa minimizar a lacuna existente entre CIM e PIM, de forma que foi elaborado neste trabalho uma proposta de análise e interpretação dos documentos de teste que possibilite a criação de um formulário de dados que descreva o problema a partir de uma sintaxe definida. Dessa forma, nesta seção serão discutidas as seguintes atividades descritas na raia CIM da Figura 11: Análise do documento de testes, definição de uma sintaxe para casos de teste e por fim a transcrição dos casos e suítes de testes utilizando a sintaxe proposta.

4.1 Análise do documento de testes

Esta proposta inicia-se com a realização de uma análise detalhada sobre os documentos de teste, os quais são documento com características independentes de computação (CIM). Mediante a dificuldade observada para a realização de transformação do documento de testes em diagramas PIM do MDA, foi estruturado um processo de interpretação e análise desse documento de forma a possibilitar a transformação automática de modelos CIM para modelos PIM.

Essa análise consiste de uma avaliação de todos os casos de teste contidos no documento um a um, selecionando aqueles que possuem maior possibilidade/viabilidade de automação conforme revisão bibliográfica e restrições técnicas ou materiais, e caso seja possível ou necessário, os casos de teste que possuírem passos em comum ou

comportamentos correlatos serão agrupados para serem executados em um mesmo *script* de automação, ou serão reunidos em uma suíte de testes comum.

Os testes selecionados para automação deverão então ser analisados e as seguintes informações deverão ser identificadas:

- Quais as entradas fornecidas a um SUT e os respectivos componentes responsáveis por realizar essas interações;
- Quais as saídas ou comportamentos esperados em resposta a uma entrada e os componentes responsáveis por realizar esses comportamentos.

Uma vez identificados esses dados, os mesmos deverão ser organizados e utilizados para o preenchimento de um formulário que contenha além dessas, outras informações importantes como os estados possíveis para cada componente e eventuais mensagens ou parâmetros que poderão ser fornecidos ao SUT. O modelo do formulário proposto encontra-se no anexo 1 deste trabalho. Por fim, nesta fase de análise dos dados contidos no caderno de testes, é necessário ainda que sejam identificados dispositivos de apoio a testes necessários para efetivar a realização dos testes. Todos esses dados deverão ser levantados, analisados e organizados na forma de um formulário que deverá abranger os seguintes campos os quais deverão ser preenchidos com as informações citadas:

- COMPONENTS IN: Componentes responsáveis por fornecer estímulos ao SUT. Ex: botões, chaves, sensores, dentre outros;
- BEHAVIOR IN: Comportamentos apresentados ou realizados por componentes de entrada. Ex: ligar/desligar uma chave, realizar a leitura de uma grandeza física por um sensor;
- STATES IN: Estados que um determinado componente de entrada pode assumir durante seu funcionamento. Ex: estados ligado ou desligado para uma chave;
- PARAMETERS: Possíveis informações ou mensagens que possam ser transmitidos ao SUT por um dispositivo de entrada. Ex: valor de uma grandeza física capturada por um sensor;
- COMPONENTS OUT: Componentes responsáveis por realizar um comportamento em resposta a uma determinada entrada. Ex: Luzes, motores, visores digitais e demais atuadores;

- BEHAVIOR OUT: Comportamento apresentado ou realizado por um componente de saída. Ex: acender ou apagar um luz, acionar um motor, mostrar um determinado dado em um visor digital;
- STATES OUT: Estados que um determinado componente de saída pode assumir. Ex: estados aceso ou apagado para uma luz, estados ligado ou desligado para um motor, diversas mensagens que podem ser apresentadas em um visor
- COMPONENTS CAP: Componentes responsáveis por capturar um comportamento apresentado pelo SUT. Ex: uma câmera para capturar imagens ou vídeos do SUT, um microfone para capturar sons emitidos pelo SUT, multímetros para capturar grandezas elétricas emitidas pelo SUT, etc.
- BEHAVIOR CAP: Comportamentos ou ações que podem ser realizadas por um componente de captura. Ex: capturar e processar imagens e vídeos do SUT, Capturar e processar sons advindos do SUT, etc.
- TEST CASES: Nomes atribuídos a cada um dos casos teste existente e descrito no documento de teste.

De acordo com a organização dos campos propostos, os componentes poderão ser de três tipos: Componentes que fornecem algum tipo de informação ou estímulo ao SUT, que serão listados no campo COMPONENTS IN do formulário; Componentes responsáveis por realizar algum comportamento em resposta a uma entrada, que serão listados na coluna COMPONENTS OUT e; Componentes responsáveis por capturar o comportamento apresentado pelo SUT físico, os quais serão listados na coluna COMPONENTS CAP.

Sobre os componentes responsáveis pela captura dos comportamentos do SUT, faz-se necessária uma avaliação mais detalhada a respeito da forma como será realizada tal captura assim como a definição dos dispositivos que serão utilizados para isso, visto que essas informações não estarão explícitas no caderno de testes.

Uma vez que os comportamentos apresentados por um sistema embarcado podem ser bastante variados de acordo com seu contexto de aplicação, a captura e processamento/extração de informações sobre esse comportamento pode, em alguns casos, se tornar uma tarefa que envolva atividades resultantes da interação de avanços em outras áreas da computação como inteligência artificial, processamento de sinais, processamento digital de imagens entre outras. Assim, é importante ressaltar que a escolha da forma como será feita a captura do comportamento do componente em teste assim como quais dispositivos serão

utilizados para isso é uma escolha que deve ser pensada e analisada pela equipe de automação, levando em conta fatores como complexidade de implementação, materiais disponíveis e custos associados.

Além disso, outras informações importantes que deverão ser elencadas no formulário são as relacionadas ao comportamento ou ação de cada componente. O comportamento associado aos componentes que fornecem um estímulo ao SUT deverão ser listados no campo BEHAVIOR IN do formulário. Os comportamentos relacionados aos componentes que realizam uma ação em resposta a uma entrada no SUT deverão ser listados na coluna BEHAVIOR OUT do mesmo formulário. Por último, deverão também ser listados os comportamentos esperados para os componentes de captura do sistema no campo BEHAVIOR CAP.

Por fim, informações a respeito dos estados que um determinado componente possa assumir assim como possíveis mensagens ou parâmetros que possam ser enviadas ao SUT por algum meio ou protocolo de comunicação também deverão ser identificadas e listadas nas colunas STATES IN, STATES OUT e PARAMETERS respectivamente.

4.1.2 Classificação dos objetos

Uma vez realizado o preenchimento prévio do formulário, o mesmo passará então por um processo de refinamento no qual os dados relacionados nos campos COMPONENTS IN, COMPONENTS OUT e COMPONENT CAP serão classificados de acordo com características e comportamentos comuns. É importante ressaltar aqui que esses dados, representam futuros objetos que farão parte do sistema de automação, de forma que essa classificação é de suma importância uma vez que a partir dela será possível definir quais classes poderão ser criadas dentro do diagrama de classes do sistema de automação de testes.

Definidas as classificações dos componentes necessários à modelagem da ferramenta de automação de testes, as classes estabelecidas deverão ser listadas nos campos CLASSES IN, CLASSES OUT e CLASSES CAP do formulário.

4.1.3 Definição de uma sintaxe para casos de teste

Após o preenchimento do formulário, a próxima atividade proposta pelo processo descrito na Figura 11 está relacionada com a geração de um documento que obedeça a uma sintaxe formal e bem definida utilizando uma gramática descrita a partir da meta-sintaxe BNF - *Backus-Naur Form*[1]. Neste trabalho, o BNF foi utilizado com o intuito de criar de uma notação formal para descrever as instruções presentes nos casos de teste de um documento do testes. Assim, foi criada então uma linguagem com sintaxe e derivações definidas para ser aplicada ao processo.

Essa linguagem tem como intuito, disponibilizar meios para transformar as descrições textuais em linguagem natural presente no documento de testes em descrições mais precisas, objetivas e padronizadas, uma vez que os documentos de teste podem ser escritos de formas diferentes por cada equipe ou pessoa responsável. Além disso, essa escrita padronizada e por meio de uma sintaxe definida possibilita ainda o mapeamento dos modelos CIM viabilizando assim a aplicação das transformações MDA em nível CIM para PIM. Dessa forma é importante ressaltar que apesar dessa diversidade de cadernos, é premissa básica para essa proposta que todas as informações necessárias para o preenchimento do formulário estabelecido na seção anterior estejam presentes.

A princípio será apresentada na Figura 12, o esquema estrutural no qual a sintaxe BNF proposta será organizada.

---

[1] A Backus-Naur Form é uma metasintaxe usada para expressar gramáticas livres de contexto, frequentemente utilizada para descrever sintaxes de linguagens de programação ou conjunto de instruções.

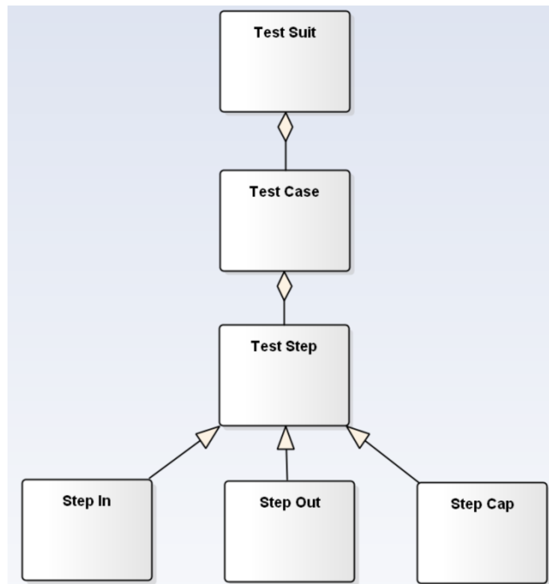

Figura 12: Estrutura inicial para formalização da sintaxe BNF.

A partir da Figura 12, é possível perceber a existência de uma hierarquia organizacional entre elementos que constituem uma estrutura básica de testes. No primeiro nível (nível mais baixo), encontram-se os elementos que compõem cada passo de um caso de teste, *Step In, Step Cap* e *Step Out*, sendo que o *Step In* descreve uma entrada que deve ser fornecida a um software ou sistema em teste, o *Step Cap* uma forma ou meio de captura de informações ou estado de um componente presente no SUT físico e o *Step Out*, uma resposta esperada ao estímulo ou entrada descrito no *Step In*. Todo passo que será realizado em um caso de teste é composto por esses três elementos básicos, ou alternativamente podem ocorrer situações na qual um desses elementos esteja ausente.

No segundo nível da Figura 12, estão os *TestCases*. *TestCases* são casos de teste compostos por um ou mais *TestSteps* que devem ser realizados sequencialmente, ou caso necessário paralelamente. Basicamente, um *TestCase* descreve um cenário onde é realizada uma execução controlada de uma dada funcionalidade do software ou SUT.

O terceiro e último nível hierárquico da Figura 12, é composto por *TestSuites*. Uma *TestSuit* é um agrupamento de *TestCases*, que proporciona uma maior organização para o desenvolvimento dos processos de automação, uma vez que tem como maior objetivo agrupar casos de teste correlacionados, possibilitando assim uma melhor organização para planejamento, gerenciamento, avaliação e execução dos *scripts* de automação.

A definição detalhada da sintaxe BNF adotada nesse estudo é mostrada a seguir e as regras e notações utilizadas foram baseadas nas mesmas regras utilizadas pela OMG na documentação da especificação da UML presentes no Anexo 2 deste trabalho [70]:

<*Test Suit*> ::= **<*Test Suit Name*> ‘ :** TestSuit’ ‘ { ’ (**<*Test Case*>** ‘;’) * ‘ } ;’
<*Test Case*> ::= **<*Test Case Name*> ‘:’ <*Test Case Class*>** ‘{ ’**<*Test Setup*>** ‘;’ (**<*Test Step*>** ‘;’) * ‘ };’
<*Test Setup*> ::= ‘ Setup: { ’ (‘(‘ (**<*Component In Name*> <*Component In Class*>** [‘ : state = ’**<*State*>**]) * ‘)’ | (‘ (’**<*Component Out Name*> <*Component Out Class*>** [‘ : state = ’**<*State*>**]) * ‘)’| ‘ (’(**<*Component Cap Name*> <*Component Cap Class*>** [‘ : state = ’**<*State*>**] ‘)’) * ‘ };’
<*Test Step*> ::= **<*Step Name*>** ‘: {’ [(‘(’ **<*Step In*>** ’)’) *] [(‘ : (’ **<*Step Out*>** ‘)’) *] [(‘ : (’ **<*Step Cap*>** ‘)’)*] ‘: (’ **<*Assert*>** ’) ;’
<*Step In*> ::= **<*Behavior In*>** **‘(‘**[**< *Parameter* >** *] **’)’** ‘ , ’ **<*Component In Name* >** ‘;’
<Step Cap> ::= **<*Behavior Cap*>** ‘(’[(**< *Parameter* >**) *] ’)’ ‘ , ’ **<*Component Cap* Name>** ‘;’
<*Step Out*> ::= **<*Behavior Out*>** **‘(’** [(**< *Parameter* >**) *] ’)’ ‘ , ’ **<*Component Out* Name>** [‘ : ’ **<*State*>**]‘;’
<***Assert***> ::= ‘{ assert({ ’ <***Component Out Name***> ‘ }, {’ <***Component Cap Name***> ‘ }), ’ <***Test Case Name***> ‘ };’
<***Behavior In***> ::= Comportamentos apresentados pelos componentes responsáveis por fornecer entradas para o SUT.
<***Component In*** Name> ::= O nome de um componente responsável por fornecer algum estímulo ao SUT.
<***Behavior Cap***> :: = Comportamentos apresentados por Componentes de captura de comportamento do SUT.
<***Component Cap Name*** > ::= O nome dos componentes responsáveis por capturar algum comportamento apresentado pelo SUT.
<***Behavior Out***> ::= Comportamentos apresentados por um Component_Out que representem o comportamento de um elemento do SUT.
<**Componen*t Out*** > ::= Representação dos componentes do SUT, o qual deseja-se testar.
<***Parameter***> ::= Mensagens ou parâmetros a serem passados para a execução de um comportamento.

<***State***> ::= Representação dos estados apresentados por um dado componente, preferencialmente escrito em letras maiúsculas.

<***Component In Class***> ::= Representação de classes de components IN.

<**Com*ponent Out Cla*ss**> ::= Representação de classes de components OUT.

<**Comp*onent Cap Cla*ss**> ::= Representação de classes de components CAP.

<***Test Case Class***> ::= Representação de classes de Test Cases.

<**Test *Suit Name***> ::= < ***Character*** > *(*<***Character***> | <***Number***>*)**

*<**Test Case Name**> ::=* <***Character***> *(*<***Character***> | <***N*umber**>)*

<***Step Name***> ::= <**Character**> (<***Character***> | <***Number***>)*

<***Character***> ::= { a, b, c, d, e, f, g, h, i, j, k, l ,m, n, o, p, q, r, s, t, u, v, w, y, z, A, B, C, D, E, F, G, H, I, J, K, L ,M, N, O, P, Q, R, S, T, U, V, W, Y, Z}

<***Number***> ::= { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

É importante ressaltar que as *tags* <***Component In Name***>, <***Component Ou Name***>, <***Component Cap Name***>, <***Behavior In***>, <***Behavior Out***>, <***Behavior Cap***>, <**Parameter**>, <***State***> <***Component In Class***>, <***Component Out Class***>, <***Compoent Cap Name***> e <***Test Case Class***> são elementos terminais mas não tiveram seus valores especificados na definição da linguagem, sendo apenas descrito o tipo de informação associada a elas. Essa característica se deve ao fato de que os valores possíveis para cada um desses elementos terminais são variáveis em função do contexto e principalmente das características do SUT em questão. Dessa forma, foi definido que os valores que cada um desses elementos terminais serão derivados dos dados preenchidos no formulário no passo anterior, para o qual será realizado o seguinte mapeamento:

- Os elementos listados no campo COMPONENTS IN do formulário fornecerão os possíveis valores para o elemento terminal <***Component In Name***> da sintaxe;
- Os elementos listados no campo BEHAVIOR IN do formulário fornecerão os valores possíveis para o elemento terminal <***Behavior In***> da sintaxe;
- Os elementos listados no campo COMPONENTS OUT do formulário fornecerão os valores possíveis para o elemento terminal <***Component Out Name***> da sintaxe;

- Os elementos listados no campo BEHAVIOR OUT do formulário fornecerão os valores possíveis para o elemento terminal <***Behavior Out***> da sintaxe;
- Os elementos listados no campo COMPONENTS CAP do formulário fornecerão os valores possíveis para o elemento terminal <***Component Cap Name***> da sintaxe;
- Os elementos listados no campo BEHAVIOR CAP do formulário fornecerão os valores possíveis para o elemento terminal <***Behavior Cap***> da sintaxe;
- Os elementos listados no campo PARAMETERS do formulário fornecerão os valores possíveis para o elemento terminal <***Parameter***> da sintaxe;
- Os elementos listados nos campos STATES IN e STATES_OUT do formulário fornecerão os valores possíveis para o elemento terminal <***State***> da sintaxe.
- Os elementos listados no campo CLASSE IN do formulário fornecerão os valores possíveis para o elemento terminal < ***Component In Class***> da sintaxe.
- Os elementos listados no campo CLASSE OUT do formulário fornecerão os valores possíveis para o elemento terminal < ***Component Out Class***> da sintaxe.
- Os elementos listados no campo CLASSE CAP do formulário fornecerão os valores possíveis para o elemento terminal < ***Component Cap Class***> da sintaxe.
- Os elementos listados no campo CLASSE TESTCASE do formulário fornecerão os valores possíveis para o elemento terminal <***Test Case Class***> da sintaxe.

Assim, de posse desse mapeamento e dos dados contidos no formulário, deverá ser feita a complementação da sintaxe BNF com os valores possíveis para os elementos terminais.

4.1.4 Transcrição dos casos de teste

Após a finalização da sintaxe e complementação dos valores dos elementos terminais de acordo com o contexto e características do SUT, essa sintaxe será utilizada para transcrição das suítes e casos de teste presentes no caderno de testes. É importante ressaltar que essa reescrita do caderno de testes, utilizando-se uma sintaxe definida, tem como principal objetivo a padronização das informações presentes no caderno, uma vez que a forma como um caderno de testes apresenta as informações sobre os testes pode variar de forma significativa de um projeto para outro, ou mesmo de acordo com o profissional responsável pela escrita do mesmo. Assim, ao aplicar-se a sintaxe definida neste trabalho, busca-se elaborar uma versão do documento de testes que contenha as informações necessárias ao processo de automação mantendo um padrão único de escrita e organização.

Dessa forma, serão realizadas três subatividades nessa etapa, sendo elas:

- Os casos de teste serão extraídos do caderno de testes um a um e serão transformados de descrição textual em linguagem natural para uma descrição formal a partir da sintaxe definida;
- Além da transcrição dos passos explicitamente presentes no caderno de testes, serão adicionados aos casos de teste as operações relativas aos processos de captura dos comportamentos do SUT;
- Caso seja necessário, os casos de teste acima gerados podem ser agrupados em uma mesma suíte de testes, com o intuito de automatizar sua execução.

Por fim vale ressaltar o fato que todo caso de teste deverá estar obrigatoriamente dentro de uma suíte de testes, mesmo que essa suíte de testes contenha um único caso de testes.

4.2 Modelo Independente de Plataforma - PIM

Os modelos PIM dentro da proposta MDA são modelos que passam a agregar informações computacionais aos modelos CIM. Tendo em vista a proposta da OMG para o MDA, que é baseada na estrutura de metamodelagem fundamentada no MOF, foi adotada a UML como linguagem de modelagem dentro deste estudo. Dessa forma, a modelagem do PIM será feita por meio dos diagramas de classes e sequência da UML, por

serem dois dos diagramas mais usualmente utilizados na modelagem de sistemas, além de possuírem uma associatividade natural entre si. Além disso o diagrama de sequência possibilita uma descrição de comportamentos em uma linha temporal, representando a invocação de métodos entre objetos de um programa. Desta forma, esta seção é dedicada à explanação das atividades envolvidas na criação dos referidos diagramas.

4.2.1 Criação do Diagrama de Classes

Para o desenvolvimento do diagrama de classes que modelarão as entidades da ferramenta automatizada de testes funcionais, é necessário que antes sejam feitas algumas considerações a respeito da estrutura e propósito desse diagrama. Serão criadas classes que se enquadrem em três frentes de atuação para o sistema de automação, sendo a primeira frente na qual as classes representarão os componentes a serem testados no SUT, ou seja, modelos que representem o dispositivo a ser testado, uma segunda frente, de classes que farão a modelagem de outros equipamentos e dispositivos para auxílio ao teste, principalmente no que tange à interação e à captura do comportamento/estado apresentado pelo SUT físico e por fim uma terceira frente na qual serão criadas classes responsáveis por modelar e executar as suítes e casos de teste. A estrutura base do diagrama é apresentada na Figura 13. As novas classes que forem inseridas a essa estrutura de classes o farão exclusivamente por meio de heranças, implementações e sobrecarga de operações , de forma que a estrutura atue como um *Framework*, no qual funcionalidades básicas necessárias à execução dos testes já foram implementadas.

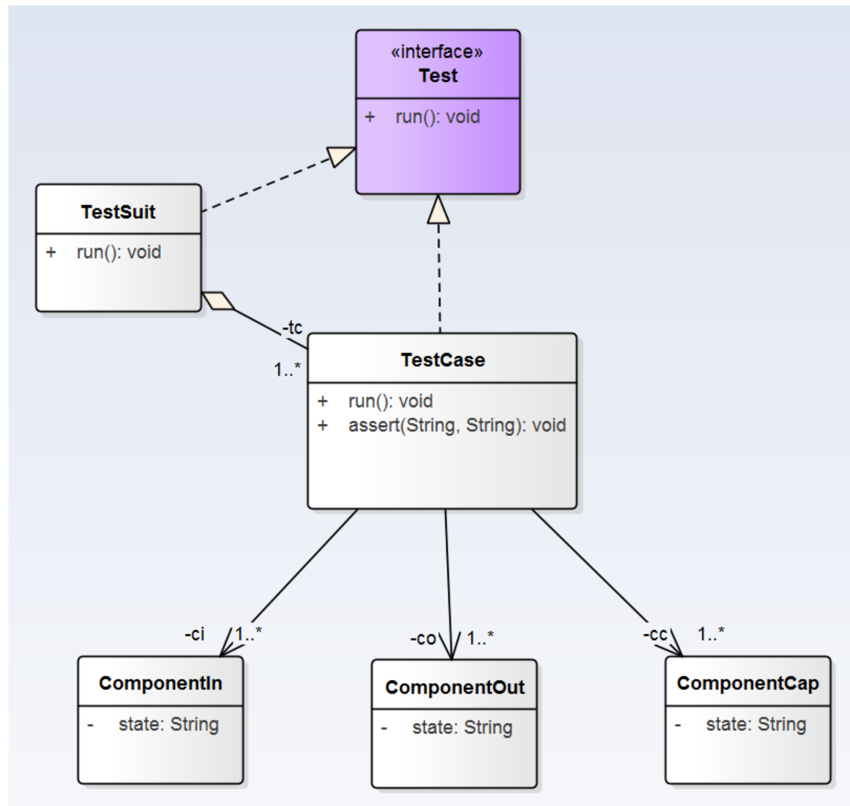

Figura 13: Estrutura inicial do diagrama de classes.

Além disso, outro ponto relevante para a automação dos testes é a captura do comportamento do sistema embarcado. O comportamento do sistema embarcado varia conforme sua aplicação e propósito, podendo assumir uma vasta gama de possibilidades, como por exemplo o acender de uma luz indicativa, uma mensagem em um *display* LCD, variações em ponteiros analógicos, acionamento de atuadores diversos entre outras. Dessa forma, é de grande importância que o comportamento do sistema embarcado tenha sido plenamente compreendido nas atividades anteriores do processo.

Essas considerações são importantes para a modelagem dos dispositivos de apoio ao teste, uma vez que a modelagem dos componentes e classes que farão o envio de informações e dados ao sistema embarcado, devem fazê-la da forma como o sistema foi concebido para entender, assim como os componentes e classes de captura do comportamento do sistema embarcado devem ser modelados de acordo com as características comportamentais do sistema embarcado. Por exemplo, um sistema embarcado que receba informações do ambiente no qual está inserido por meio de sensores de luminosidade e que ligue ou desligue uma fonte de luz artificial de acordo com a intensidade luminosa captada, precisa receber as informações de luminosidade de acordo com as especificações do sensor,

por exemplo um sinal analógico de 4 a 20 mA. Da mesma forma, para capturar o comportamento de acender ou apagar a fonte de luz artificial, pode-se utilizar um dispositivo de captura de imagens como uma câmera e posteriormente aplicar sobre a imagem capturada um algoritmo de processamento digital de imagens para extrair as informações desejadas dessa imagem.

Feitas essas considerações, as próximas seções abordarão como realizar a modelagem das classes da ferramenta de forma a atender essas características, dividindo as classes em quatro categorias: Classes que modelem componentes do SUT; Classes que modelem componentes responsáveis por interagir com o SUT por meio de envio de entradas; Classes responsáveis por capturar o comportamento apresentado pelo SUT físico e; Classes responsáveis por modelar o comportamento desejado para os casos de teste.

3.2.1.1 Definição das classes Component Out

De posse do formulário contendo todos os dados e informações levantadas na fase de análise, as classes que representam os elementos constituintes do SUT e seus comportamentos serão modeladas. O primeiro passo é criar as classes que representam os elementos que compõem o SUT. Para isso serão utilizadas as informações contidas nos campos CLASSES OUT e BEHAVIOR OUT presentes no formulário. Para cada elemento listado no campo CLASSES OUT do formulário será criada uma classe que deverá herdar da super-classe *ComponenteOut* já previamente definida no diagrama de classes, e cada elemento listado no campo BEHAVIOR OUT será mapeado para um método dessa classe.

Em seguida, o próximo passo é estabelecer os estados que o componente poderá assumir de acordo com o comportamento esperado e descrito no campo STATES OUT do formulário. Uma vez identificados os estados possíveis para cada componente, os mesmos serão enumerados dentro de uma classe *Enumeration*, que deverá ser nomeada a partir do nome da classe a qual o *Enumeration* descreve os estados sucedida da palavra "*states*".

4.2.1.2 Definição das classes Component In

A classe *ComponentIn* definida no diagrama de classes da Figura 13 atua como a super-classe de todas as classes responsáveis pelo envio de estímulos e/ou informações ao SUT. No formulário gerado a partir da análise do caderno de testes, essas interações são apresentadas conjuntamente pelos campos COMPONENTS IN e BEHAVIOR IN. Dessa forma, essas classes serão modeladas baseadas nos componentes descritos no

campo CLASSES IN do formulário. Ou seja, para cada elemento presente nesse campo, será criada uma classe de mesmo nome no diagrama de classes. Além disso, os comportamentos descritos no campo BEHAVIOR IN do formulário, serão mapeados na forma de métodos dessas classes. Por fim, de forma semelhante ao que foi realizado nas classes *ComponentOut*, deverão ser criadas classes *Enumeration* que guardem os estados possíveis para cada uma dessas classes, sendo que esses estados estarão listados na coluna STATES IN do formulário.

4.2.1.3 Definição das classes Component Cap

A classe *ComponentCap* do diagrama de classes, representa a super-classe de todas as classes responsáveis por de alguma forma capturar os comportamentos funcionais do SUT. Assim, as classes filhas de *ComponentCap* serão responsáveis por capturar o comportamento apresentado pelo SUT, e por meios computacionais extrair informações e dados sobre o comportamento apresentado, de forma que, posteriormente possa ser feita a avaliação desse comportamento em relação ao comportamento esperado para o componente em teste.

Além disso, em muitos casos será necessária a utilização de equipamentos específicos para capturar tais comportamentos, como câmeras para captura visual, microfones para captura sonora ou outros instrumentos para captura de outras grandezas físicas. Cada equipamento necessário deverá estar presente no formulário no campo COMPONENTE CAP, sendo cada um dos elementos presentes mapeado para uma classe que herde da classe *ComponentCap*.

Além disso, essas classes farão uso das classes *Enumeration* criadas até o momento com o intuito de determinar o estado esperado pelo componente a ser testado, o qual será posteriormente utilizado para comparação com os estado capturado e identificado para um dado componente em teste.

Apesar do procedimento até aqui apresentado ser praticamente idêntico ao processo de modelagem proposto para as demais classes já discutidas, os componentes de captura podem possuir algumas características peculiares que diferem dos demais componentes. Seguindo essa estrutura a forma de acesso aos estados permanece inalterada, no entanto é possível que cada classe especializada seja responsável por realizar os processamentos e tratamentos necessários para cada caso em particular.

4.2.1.4 Definição das classes TestCase

Por fim, serão modeladas classes responsáveis pela execução dos casos de teste. Essas classes terão quatro responsabilidades básicas, sendo elas:

1. Realizar o *setup* do teste, que consiste na instanciação dos objetos necessários para a execução de cada caso de teste;

2. Implementar método abstrato *run*() herdado diretamente da super-classe *TestCase* e executar passo a passo um caso de teste;

3. Fazer a avaliação do resultado de cada passo de um caso de teste, comparando os valores de estados esperado e obtido;

4. Gerar o resultado final da execução do teste.

Ressalta-se que para cada elemento presente no campo TEST CASE do formulário será criada uma classe filha de *TestCase* no diagrama de classes. A classe *TestCase* possui a declaração do método abstrato *run*(), responsável pela execução dos passos de um caso de teste. Assim, é obrigação de cada nova classe filha de *TestCase* inserida no diagrama, a implementação desse método. Além disso, as classes *TestCase* possuem a responsabilidade de fazer a avaliação dos estados esperados e obtidos por cada passo de teste. Essa tarefa será realizada pelo método *assert*() da mesma classe. Esse método recebe por parâmetro dois constantes do tipo *string*, realizando a comparação entre elas, e registrando o resultado para cada passo realizado.

Por fim, é importante ressaltar que as características comportamentais do método *run()* de cada uma das classes serão modeladas por diagramas de sequências os quais serão abordados na próxima sessão.

4.2.2 Diagramas de Sequência

O diagrama de sequência da UML tem como característica modelar a sequência de eventos assim como a interação entre diversos objetos por meio de troca de mensagens e invocação de métodos entre eles [40]. Em função dessas particularidades, o diagrama de sequência possui características bastante úteis para a modelagem de casos de testes, que em grande parte consistem de sequencias de atividades a serem realizadas com o intuito de verificar e validar comportamentos de um sistema.

Dessa forma o diagrama de sequência foi escolhido para modelagem comportamental dos casos de teste nesta proposta. Cada classe filha de *TestCase* modelada na

atividade anterior possui um cenário de teste associado, o qual será descrito pela implementação do um método *run()* responsável pela execução dos casos de teste. Assim, a modelagem comportamental desse método será feita por meio de um diagrama de sequência, que detém a responsabilidade de invocar o método *run(),* que por sua vez é representado na linha de vida de um objeto *TestCase* instanciado. Dentro do método *run()* serão realizados o *setup* do teste, assim como os passos de teste.

Para isso serão utilizadas as informações presentes nos casos e suítes de teste escritos utilizando-se a sintaxe BNF proposta neste trabalho. A sintaxe criada neste trabalho foi estruturada com o intuito de atender ao domínio específico de testes de software, sendo que cada elemento da sintaxe foi definido com o intuito de possibilitar uma transformação para um elemento do diagrama de sequencia. Portanto, ao utilizar a estrutura definida para a sintaxe, é possível realizar um mapeamento para que seja feita uma transformação automática dos elementos descritos no caso de testes para respectivos elementos de um diagrama de sequência.

4.2.2.1 Mapeamento e transformação de CIM para PIM

Abaixo segue uma estrutura base criada a partir da sintaxe proposta com o formato genérico contendo uma Suíte de testes com um caso de teste. Esta estrutura foi empregada para mapear a conversão da sintaxe para um Diagrama de Sequência. Esta seção foi desenvolvida com o propósito de definir o processo de transformação de CIM para PIM, sendo organizada em seis passos

1<***Test Suit Name***> : TestSuit{

2 <***Test Case Name***> : <***Test Case Class***>{

3 Setup: { (<***Component In Name***> <***Component In Class***>: state = <***State***>) (<***Component Out Name***> <***Component Out Class***>: state = <***State***>) (<***Component Cap Name***> <***Component Cap Class***>: state = <***State***>) };

4 <***Step Name***>: { (<***Behavior In***>(<***Parameter***>), <***Component In Name***>) : (<***Behavior Out***>(), <***Component Out Name***>) : (<***Behavior Cap***>(), <***Component Cap Name***>) : (assert({ <***Component Out Name***> }, { <***Component Cap Name***> }), <***Test Case Name***>) };

};

};

Passo 1: O valor referente a *tag* <***Test Suit Name***> será mapeado na forma de um objeto *controller* do diagrama de sequencia. Esse elemento é uma instância nomeado pelo próprio valor da *tag* <***Test Suit Name***> e a classe desse objeto será *TestSuit*, como apresentado na Figura 14.

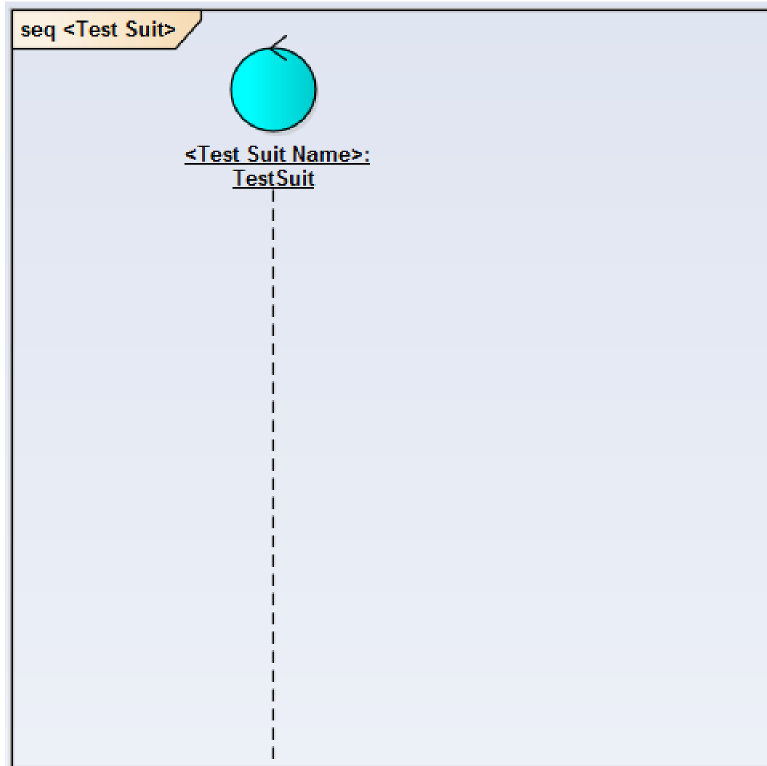

Figura 14: Criação de uma *Lifeline controller* do tipo *TestSuit*.

Passo 2: O valor referente a *tag* <***Test Case Name***> será mapeado na forma de um objeto *controller* do diagrama de sequência. Esse elemento é uma instância da classe <***Test Case Class***> nomeado pelo próprio valor da *tag* <***Test Case Name***>. É criada uma mensagem com o método *run*() do <***Test Suit Name***> em direção ao <***Test Case Name***>. Esse procedimento é repetido para cada *tag* <***Test Case Name*** > existente dentro escopo da *tag* <***Test Suit Name***>, ignorando o conteúdo existente entre os elementos "{" e "};". A criação desse elemento do modelo é apresentada na Figura 15.

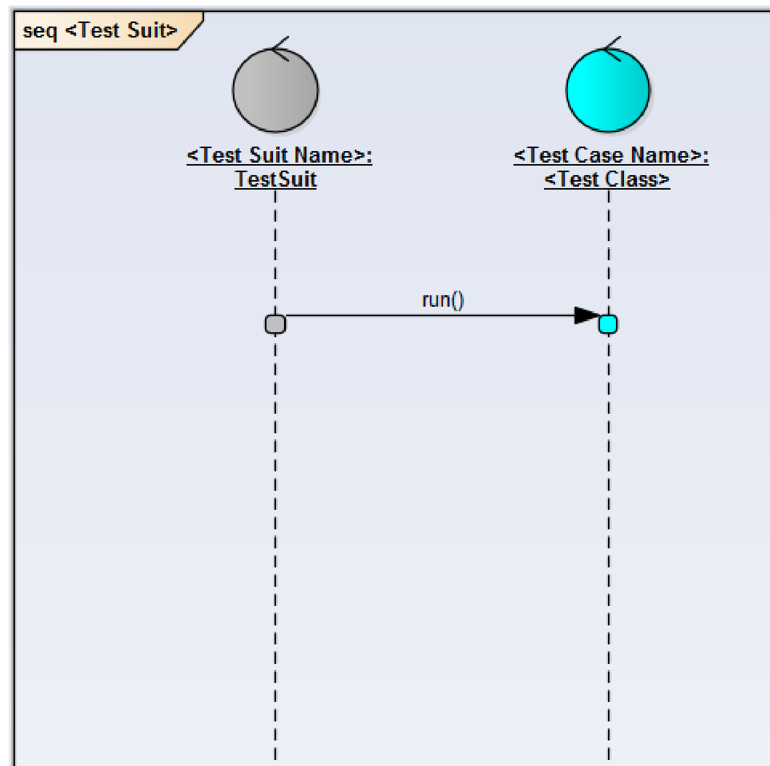

Figura 15: Invocação do método *run()* do objeto controller da classe *TesteCase* pela classe *Testsuit*.

Passo 3: Para cada *tag* <***Test Case Name***> será criado um novo diagrama de sequência nomeado pelo valor da *tag* <***Test Case Name***>, no qual será criado um *EndPoint* enviando uma mensagem *run*() para um objeto *controller* com o nome <***Test Case Name***> : <***Test Case Class***>. Esse procedimento pode ser observado na Figura 16.

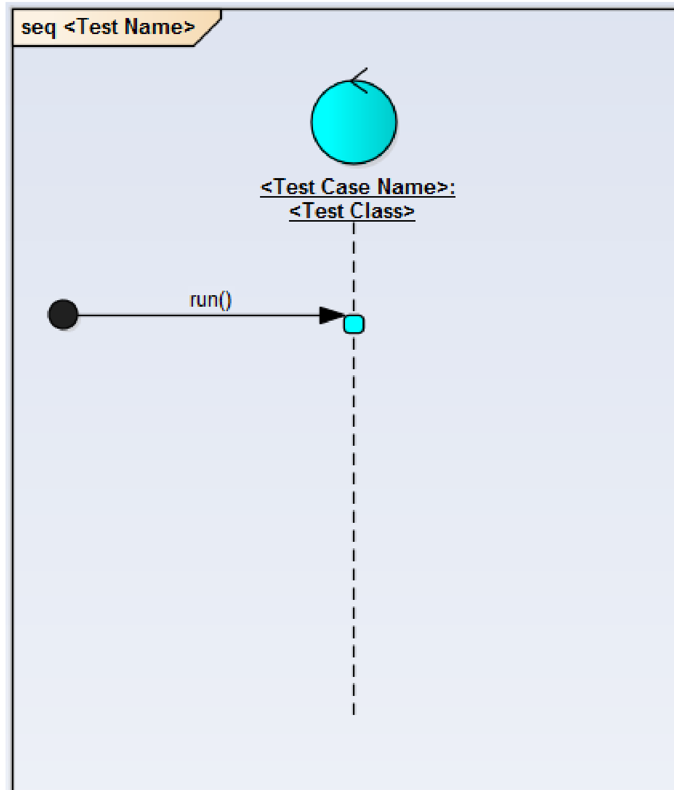

Figura 16: Criação da *Lifeline controller* da classe de TesteCase.

Passo 4: Dentro do escopo de <***Test Case Name***> : <***Test Case Class***>, ao encontrar a palavra *Setup*, serão criados os objetos necessários para a execução do teste. Cada objeto que será instanciado encontra-se entre parênteses, de forma que ao encontrar um símbolo "(", será criada uma *lifeLine* nomeada da seguinte forma: o primeiro termo refere-se ao nome do objeto a ser instanciado contido na *tag* <***Component In Name***>, <***Component Out Name***> ou <***Component Cap Name***>, o segundo termo é a classe desse objeto cujo valor está contido na *tag* <***Component In Class***>, <***Component Out Class***> ou <***Compoent Cap Class***>. Além disso, uma mensagem com o estereótipo <<*create*>> será criada invocando o construtor da classe do objeto criado. Essa mensagem tem como ponto de partida o objeto <***Test Case Name***> : <***Test Case Class***> e vai em direção à *lifeLine* do objeto criado. Além disso, caso sejam encontrados os termos ": *state* = ", o valor da *tag* <***State***> deve ser passado como parâmetro para o construtor, caso contrário o construtor padrão é invocado. Esse passo repete-se até que seja encontrado o símbolo ";", e resultado pode ser visto na Figura 17.

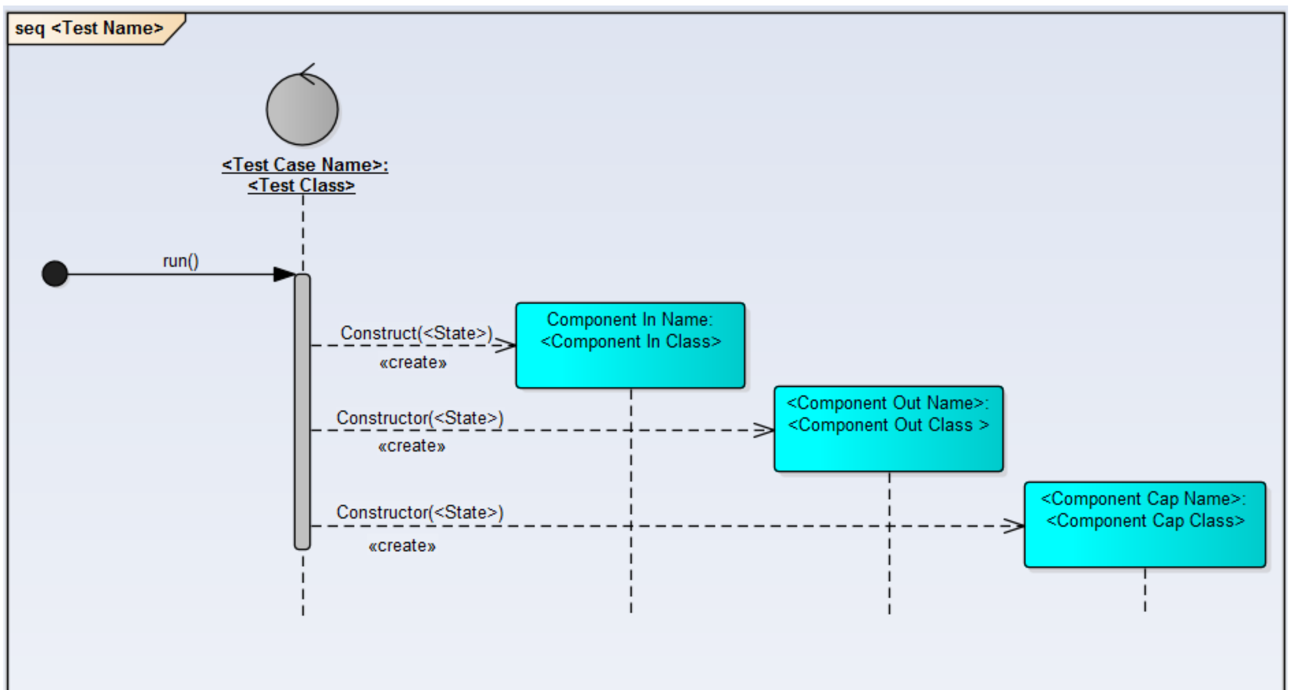

Figura 17: Acréscimo das *Lifelines* instanciadas ao digrama.

Passo 5: Após finalizar o passo 4, inicia-se então a busca por elementos com a *tag* <***Step Name***>. Para cada <***Step Name***>, busca-se o primeiro "(". Verifica-se então se o primeiro termo dentro dos parênteses é igual ao termo "*assert*". Caso sim, será criada uma auto mensagem partindo do objeto <***Test Case Name***> : <***Test Case Class***>, invocando o método *assert*, passando como parâmetros, a invocação dos métodos "*getState*()" dos objeto descritos pelas *tags* <***Component Out Name***> e <***Component Cap Name***>. Caso contrário, é criada uma mensagem invocando o método descrito na *tag* <***Behavior In***>, <***Behavior Out***> ou <***Behavior Cap***> partindo do *controller* em direção ao objeto cujo nome seja igual ao encontrado na *tag* <***Component In Name***>, <***Component Out Name***> ou <***Component Cap Name***> que estejam dentro dos parênteses. Após o fechamento dos parênteses, caso seja encontrado um símbolo ":", o processo até descrito neste item é repetido. Caso seja encontrado um símbolo ";", esse passo é finalizado e inicia-se a busca por outro elemento com a *tag* <***Step Name***>. Caso não existam outros elementos com a *tag* <***Step Name***>, significa que não existem mais passos a serem processados nessa estrutura. A Figura 18 apresenta o resultado dessa atividade.

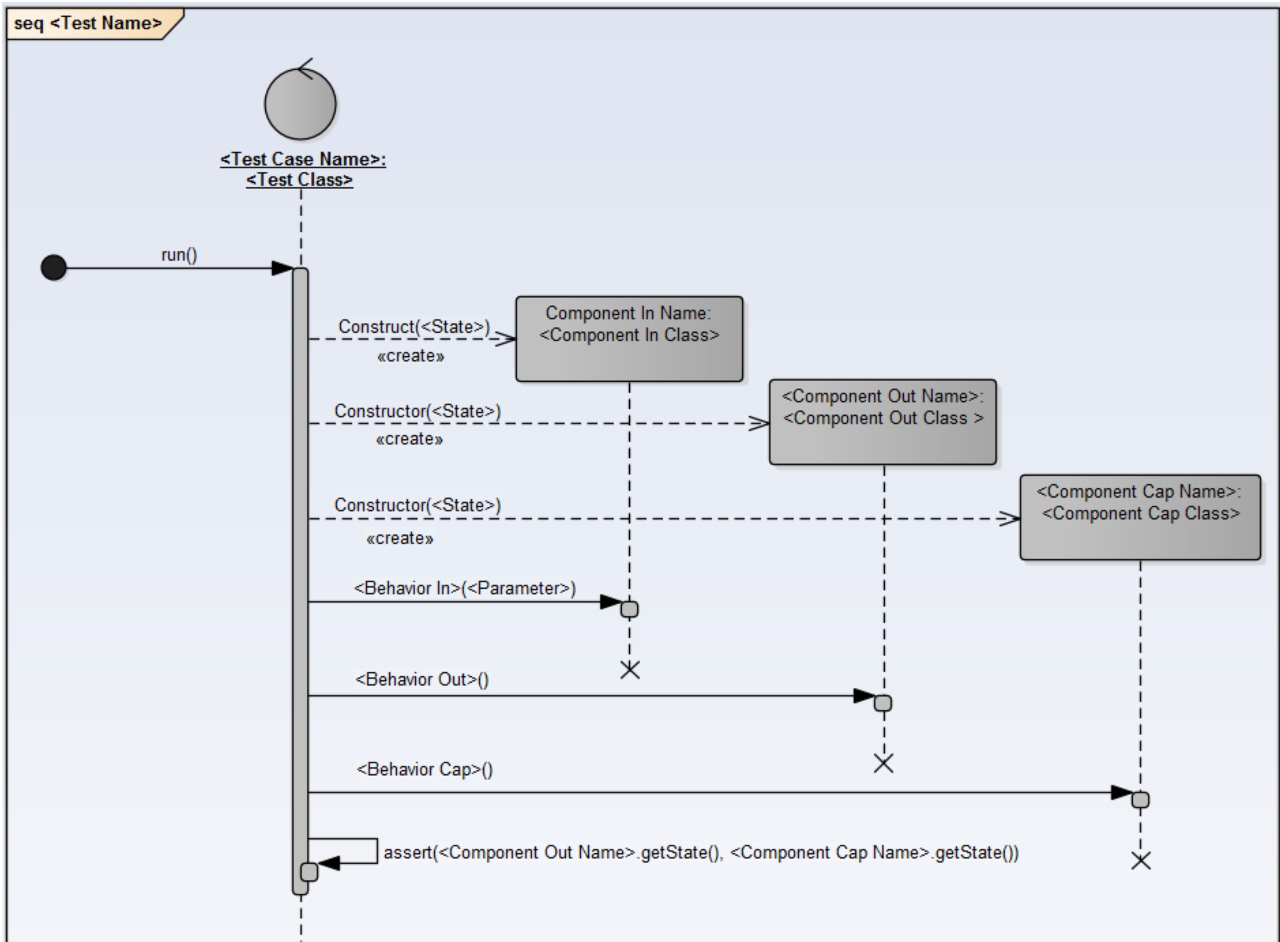

Figura 18: Esquema completo de diagrama de sequência para um caso de testes.

Passo 6: Caso existam outros elementos com a *tag* <***Test Case Name***>, os passos de 3 a 5 serão repetidos, até que não existam mais elementos com a *tag* <***Test Case Name***>.

O mapeamento descrito acima, pode ser realizado para documentos que obedeçam a uma estrutura XMI, de forma que uma ferramenta que faça esse mapeamento gerará um documento em XMI que corresponda a esse modelo em UML gerado de forma automática. Mas não é objetivo deste estudo criar uma ferramenta com esse propósito.

Concluída a modelagem dos diagramas de sequência, as atividades de modelagem propostas nesse processo estarão finalizadas. As próximas atividades do processo proposto estão relacionadas às transformações MDA para a geração do código do sistema.

4.3 Modelo Específico de Plataforma - PSM

Até o presente momento foram traçadas e explanadas atividades relacionadas à modelagem das ferramentas de automação de testes propostas neste trabalho, sendo que como resultado das atividades até aqui apresentadas, foram gerados diagramas de classes e sequência os quais descrevem a estrutura e comportamento de um sistema para execução automática de testes funcionais sob a forma de modelos PIM. Diante disso, e seguindo a proposta do MDA, nesta seção será abordada a etapa de transformações de modelos PIM para modelos PSM.

4.3.1 Aplicação de transformações MDA automatizadas nos diagramas de Classes e Sequências

Uma vez finalizada a modelagem dos diagramas propostos, os mesmos serão submetidos a transformações MDA de modelos PIM para modelos PSM. Logo, nesta etapa é necessário definir a linguagem alvo desejada para a ferramenta.

As transformações de modelos PIM para modelos PSM do MDA são realizadas mediante o mapeamento dos elementos que compõem um modelo UML ou outra linguagem baseada no meta-modelo MOF da OMG. Esse mapeamento deve estabelecer uma relação de correspondência entre elementos dos modelos alvo e fonte, sendo que a adoção de uma forma padrão para a representação dos elementos de um modelo como o XMI, possibilita além da intercambialidade de modelos entre diferentes ferramentas de modelagem e transformação, a identificação de cada elemento presente nos modelos fonte e alvo a partir da estrutura de *tags* estabelecida para a UML. Ou seja, o formato XMI de compartilhamento de modelos permite que cada elemento de um modelo seja mapeado e posteriormente transformado para diferentes plataformas. No entanto, ressalta-se que esta proposta não tem como objetivo criar o mapeamento de modelos UML para uma plataforma específica, e sim aplicar mapeamentos já existentes e utilizados em ferramentas criadas e mantidas por empresas associadas à OMG.

Diante disso, fazendo uso da estrutura estabelecida pela OMG na qual as regras necessárias para mapeamentos e transformações são definidas e fazendo uso do formato XMI que organiza todos os dados contidos em um modelo através de sua estrutura de *tags*, empresas de desenvolvimento de ferramentas de suporte ao MDA, realizam os

mapeamentos dos modelos para as plataformas desejadas. Na página da OMG na internet é possível encontrar uma lista de ferramentas e empresas parceiras que suportam o MDA, como as ferramentas CASE Enterprise Architect da empresa Sparx Systems, Rational Software Architect da IBM, OptmalJ da empresa Compuware dentre outras.

4.4 Código

Após a análise e estruturação dos modelos CIM e posteriores transformações de CIM para PIM e PIM para PSM faz-se a transformação do modelo específico de plataforma para o código fonte na linguagem especificada para o modelo PIM. Dessa forma, nesta seção são abordadas as atividades do processo proposto relacionadas a essa transformação, sendo elas: Transformação de modelo PSM para código, implementação do código comportamental das classes e por fim, a compilação e execução da ferramenta de automação de testes.

4.4.1 Transformação de PSM para Código

É importante ressaltar que dentre as classes modeladas e subetidas a transformações até o momento, aquelas que representam os componentes de entrada, saída e de captura de comportamentos do SUT, terão somente seu código estrutural gerado automaticamente. As classes *TestCase* e *TestSuit* por sua vez serão transformadas e terão seus códigos, estrutural e comportamental gerados automaticamente pelas transformações MDA. A característica comportamental de cada uma dessas classes é representada pelo método *run*(), o qual terá seu código baseado no respectivo diagrama de sequência.

Estruturalmente, o MDA apresenta através das diversas camadas de modelagem e meta modelagem definidas a partir do MOF, todas as regras e informações necessárias ao mapeamento dos elementos construtivos de um modelo para elementos de código de uma linguagem de programação. No entanto, as transformações e mapeamentos mais comuns existentes nas ferramentas de suporte ao MDA concentram-se em transformações de diagramas estáticos ou estruturais como o diagrama de classes. Transformações de diagramas comportamentais são ainda pouco suportadas nas ferramentas.

Porém, dada a característica do diagrama de sequência modelado nesta proposta e também em função do foco em um dominio especifico de testes de software no

qual todo o conteúdo presente no diagrama é exatamente o que deverá estar no codigo para execução dos teste, é possível mapear todos os elementos do diagrama de sequência em código por meio dos elementos presentes no XMI que descreve todos os elementos gráficos utilizados para modelagem do diagrama capturando suas características.

4.4.2 Implementação de código comportamental

Uma vez realizadas todas as tranformações de PSM para código propostas pelo MDA, todo o código estrutural e parte do código comportamental da ferramenta de automação de testes está pronto. No entanto, deverão ainda ser realizadas as implementações comportamentais dos métodos das classes filhas de *ComponentIn*, *ComponentOut* e *ComponentCap*. Essas implementações devem ser realizadas pela figura de um programador, visto que a modelagem dos mesmos pode acarretar em situações na qual os modelos comportamentais desses métodos tornem-se demasidamente complexos.

4.4.3 Compilação e execução do programa

Por fim, após a realização de todas as atividades previamente descritas do processo proposto, é feita então a compilação do código utilizando-se um compilador adequado para a linguagem alvo adotada e posterior execução da ferramenta responsável por executar automaticamente os casos de teste modelados.

4.5 Considerações do Capítulo

Neste capítulo foi apresentada a proposta de processo de desenvolvimento de sistemas de automação de testes funcionais voltados para sistemas embarcados e norteada pelas diretrizes propostas pela OMG para o MDA. O processo definido neste capítulo busca atender às três etapas do MDA, abordando modelos CIM, PIM e PSM, assim como estabelece formas e métodos para possibilitar as transformações automáticas entre esses modelos. No entanto, algumas limitações do processo, principalmente no que tange à sintaxe definida utilizando-se BNF dever ser mencionadas. A definição da sintaxe apresentada neste capítulo, não abrange alguns elementos como possibilidades de execução de laços de repetição, execuções condicionais ou paralelas. Estes elementos, apesar de serem bastante usuais na

definição e criação de casos de teste, não foram implementados devido ao grande aumento da árvore de possibilidades que a sintaxe necessitaria abranger. Visto que a proposta do trabalho é a de verificar a aplicação do paradigma do MDA a testes, optou-se pela definição de uma sintaxe mais simples e direta. Entretanto, ressalta-se que o MDA e suas transformações suportariam esses elementos uma vez que eles são descritos por fragmentos combinados do diagrama de sequencia da UML.

Entretanto, destaca-se o fato de que o processo proposto e descrito neste capítulo, principalmente por meio da definição da sintaxe para escrita dos casos de teste em um formato padrão, proporciona avanços no que tange à viabilização de transformações automáticas de modelos CIM para PIM. Outras propostas como a apresentada por Kriouile, Addamssiri e Gadi [69] buscaram estabelecer regras para o mapeamento de elementos CIM para elementos PIM, utilizando para isso o diagrama de casos de uso da UML. Entretanto, a proposta dos autores limitou-se a estabelecer essa relação, sem no entanto definir regras ou diretrizes para a escrita dos diagramas, tornando assim os modelos alvo das transformações suceptíveis às variações de modo e forma de escrita.

5 TESTE EXPERIMENTAL DA PROPOSTA

Neste capítulo será realizada uma experimentação da proposta deste estudo aplicada a um caso de testes. Dessa forma, o restante deste capítulo abordará os materiais e métodos necessárioas a essa experimentação assim como fará um detalhamento das atividades realizadas fazendo um paralelo com as seções e subseções apresentadas no capítulo quatro deste estudo.

5.1 Materiais e Métodos

Para a realização desta experimentação foram utilizados como materiais, um caso de testes de sistema de controle de acendimento automático de farol automotivo controlado por um sensor de luminosidade e uma chave de três posições, ligada, desligada e automático adaptado de Bringman e Kramer [28]. Esse caso de teste consiste em uma sequencia de seis passos, sendo eles:

1. <u>Colocar</u> a **chave** para posição desligada, e o **farol** também deve estar desligado;
2. <u>Mudar</u> a posição da **chave** para ligada e o **farol** deve acender;
3. <u>Retornar</u> a **chave** para posição desligada e o **farol** deve apagar;
4. <u>Mudar</u> a **chave** para posição automático e o **farol** deve permanecer apagado;
5. <u>Mudar</u> o valor do **sensor** de luminosidade para 70% e o **farol** deve permanecer apagado;
6. <u>Mudar</u> o valor do **sensor** de luminosidade para 60% e o **farol** deve acender.

Além disso, foram utilizados neste experimento um painel ou cluster automotivo como sistema em teste. Um painel automotivo é um dispositivo embarcado presente em praticamente todos os modelos de carros atuais, sendo uma interface que possibilita a transmissão de informações importantes a respeito do funcionamento do veículo para o motorista. Dentre estas informações, foi utilizado neste experimento a espia de farol presente no cluster.

Outro dispositivo utilizado foi uma câmera digital. Este dispositivo foi aplicado como meio de captura de imagens do cluster automotivo, e também os algoritmos de

processamento digital de imagens de filtro de média, conversão para tons de cinza e binarização.

Além disso, durante a fase de modelagem, será utilizada a ferramenta de suporte ao MDA Enterprise Architect® (EA). Essa ferramenta proporcionará todo o ambiente gráfico para o processo de modelagem descrito na proposta deste estudo, assim como meios de aplicar as transformações MDA aos modelos gerados.

O Enterprise Architect da empresa Sparx Systems é uma ferramenta proprietária de modelagem e projeto de software. Ela permite desde a concepção até a construção do sistema, abrangendo todo o ciclo de vida de desenvolvimento. A plataforma dá suporte a notações UML da OMG, assim como também permite a modelagem em SysML, BPMN, BPEL, SoaML, SPEM, WSDL e outros.

Essa ferramenta de modelagem permite geração automática de documentação e código fonte, suportando ainda a realização de transformações e mapeamentos entre modelos a partir do suporte ao MDA. O código gerado pode ser nas linguagens: Action Script, C, C ♯, C++, Delphi, Java, PHP, Python, Visual Basic e Visual Basic .NET. Além disso, é possível adicionar *plugins* que viabilizam a transformação dos modelos em outras linguagens.

Outra característica importante do Enterprise Architect é a rastreabilidade dos elementos dos diagramas através de matrizes de relacionamento e links entre os objetos. Dessa forma, é possível verificar a ligação entre os diversos modelos utilizados para descrever uma dada aplicação. A plataforma de modelagem também permite a padronização de nomenclaturas, controle de versionamento e engenharia reversa, com geração de modelos a partir de código, incluindo script de banco de dados. Por fim ressalta-se ainda o fato da ferramenta EA suportar o padrão XMI da OMG de compartilhamento de modelos, possibilitando a importação e exportação dos modelos gerados para documentos que obedeçam a esse padrão.

Nesta experimentação, foi utilizada ainda a linguagem de programação Java como linguagem alvo de mapeamentos e transformação, além de um compilador adequado para essa plataforma.

Por fim, outros recursos conceituais para o mapeamento de transformações como MOF, XMI e UML serão utilizados como elementos de apoio ao longo da realização desta experimentação.

O método aplicado neste experimento foi o descrito no capítulo quatro deste trabalho, incluindo todas as atividades descritas na Figura 11.

5.2 Teste Experimental

A experimentação deste estudo foi guiada pelo fluxo descrito na Figura 11 da seção 3, priorizando as atividades de uma raia por vez, ou seja, as atividades descritas nesta seção são organizadas de acordo com a separação por raias do diagrama de atividades descrito na Figura 11.

5.2.1 Modelo Independente de Computação – CIM

As primeiras atividades descritas na Figura 11 são relacionadas com modelos independentes de computação, sendo elas a análise do documento de testes, classificação de objetos, definição de uma sintaxe para escrita de casos de teste e a transcrição dos casos de teste para a sintaxe proposta.

5.2.1.1 Análise do documento de testes

A partir do caso de teste utilizado como exemplo para essa experimentação, foram retiradas as informações descritas na seção 4.1.1 deste trabalho sendo posteriormente realizado o preenchimento do formulário de dados.

Primeiramente foram identificados os componentes presentes no caso de testes que atuam como meios de entrada de dados e estímulos ao SUT, assim como os comportamentos, estados e parâmetros associados aos mesmos. Diante desta análise foram identificadas uma chave de três posições e um sensor de luminosidade, de forma que estes componentes foram listados no campo COMPONENTS IN do formulário. Além disso, os comportamentos ligar chave, desligar uma chave, colocar a chave na posição automático, alterar o valor do sensor de luminosidade para 70% e alterar o valor do sensor de luminosidade para 60% também foram identificados e listados no campo BEHAVIOR IN do mesmo formulário. Diante destes comportamentos, os estados possíveis para cada camponente e os parametros necessários para os mesmos foram determinados e listados nos campo STATES IN e PARAMETERS do formulário.

Após a identificação dos componentes de entradas para o SUT, foi identificado também o componente farol que atua como meio de saída para o SUT, assim como seus comportamentos acender e apagar e os estados aceso e apagado. Esses dados foram

listados respectivamente nos campos COMPONENTS OUT, BEHAVIOR OUT e STATES OUT do formulário.

Em seguida, após a análise do caso de teste em questão, foi identificado que a avaliação dos comportamentos apresentados pelo SUT são realizados essencialmente de forma visual (verificar a condição aceso ou apagado do farol), de forma que optou-se pelo uso de uma câmera que pudesse capturar fotos do farol e que por meios computacionais fosse criado um algoritmo de processamento digital de imagens que pudesse identificar a condição aceso ou apagado do mesmo. Dessa forma um componente câmera foi listado no campo COMPONENTS CAP do formulário, e os comportamentos de capturar imagem e processar imagem foram definidos para esse componente e listados no campo BEHAVIOR CAP.

Por fim, foi criado um nome que descreva o caso de testes em questão, e esse nome foi escrito no campo TEST CASE NAME do formulário.

Abaixo segue o formulário preenchido.

Tabela 3: Formulário de dados de teste.

FORMULÁRIO DE DADOS DE TESTE **Caso de Teste:** TestCaseAutomaticLight
LISTAS DE COMPONENTES
COMPONENTS IN: *lightSwitch; luminositySensor*
BEHAVIOR IN: *SwitchOn; switchOff; switchAuto; setLuminosity*
STATES IN: *On; Off; Auto;* 100%; 70%; 60%
PARAMETERS: 100; 70; 60
COMPONENTS OUT: *headLight*
BEHAVIOR OUT: *turnOn; turnOff*
STATES OUT: *On; Off*
COMPONENTS CAP: *camera*
BEHAVIOR CAP: *captureImage; processImage*

5.2.1.2 Classificação dos objetos

A atividade seguinte do processo de desenvolvimento de sistemas de automação de teste proposto, visa realizar uma classificação dos elementos listados nos campos do formulário até o momento. Dada a simplicidade do caso em estudo, não mais que um componente de cada tipo foi listado no formulário, de forma que as classificações resultaram em quatro classes que descrevem cada uma, um tipo de componente listados no formulário. Como resultado dessa atividade, foram identificadas as classes *Switch*, derivada do componente Chave de tres posições; a classe *Light* derivada do componente farol;s classe Sensor derivada do componente sensor de luminosidade e; a classe *Camera*, derivada do componente *camera*.

A seguir é mostrado o formulário completo com os novos dados identificados nesta etapa do processo.

Tabela 4: Formulário de dados com a classificação de componentes.

FORMULÁRIO DE DADOS DE TESTE **Caso de Teste:** TestCaseAutomaticLight
LISTAS DE COMPONENTES
CLASSE IN: *Switch; Sensor*
COMPONENTS IN: *lightSwitch; luminositySensor*
BEHAVIOR IN: *SwitchOn; switchOff; switchAuto; setLuminosity*
STATES IN: *On; Off; Auto*; 100%; 70%; 60%
PARAMETERS: 100; 70; 60
CLASSES OUT: *Light*
COMPONENTS OUT: *headLight*
BEHAVIOR OUT: *turnOn; turnOff*
STATES OUT: *On; Off*

CLASSES CAP: *Camera*
COMPONENTS CAP: *camera*
BEHAVIOR CAP: *captureImage; processImage*

5.2.1.3 Definição de uma sintaxe para os casos de teste

Conforme descrito na sação 4.1.2 deste trabalho, e fazendo uso das informações organizadas no formulário de dados, foi feita a complementação dos elementos terminais da sintaxe utilizando o BNF. Cada elementos listado no formulário de dados foi mapeado como valores terminais na definição da sintaxe, utilizando a própria sintaxe para determinar a forma de escrita dos mesmos.

Os elementos ***lightSwitch*** e ***luminositySensor*** listados no campo COMPONENTS IN do formulário, foram mapeados como os valores terminais da *tag* <***Component In Name***> da sintaxe, de maneira que a forma final dessa *tag* é: <***Component In Name***> ::= { '*lightSwitch'* | '*luminositySensor'* };

Os elementos ***SwitchOn, switchOff, switchAuto*** e ***setLuminosity*** listados no campo BEHAVIOR IN do formulário foram mapeados como valores terminais da *tag* <***Behavior In***> da sintaxe, de maneira que a forma final dessa *tag* é: <***Behavior In***> ::= { '*switchOn'* | 'switchOff' | '*switchAuto'* | '*setLuminosity'* };

O elemento ***headLight*** listado no campo COMPONENTS OUT do formulário foi mapeado como valor terminal da *tag* <***Component Out Name***> da sintaxe, de maneira que a forma final dessa *tag* é: <***Component Out Name***> ::= { '*headLight'* };

Os elementos ***turnOn*** e ***turnOff*** listados no campo BEHAVIOR OUT do formulário foram mapeados como valores terminais da *tag* <***Behavior Out***> da sintaxe, de maneira que a forma final dessa *tag* é: <***Behavior Out***> ::= { '*turnOn'* | '*turnOff'* };

O elemento ***camera*** listado no campo COMPONENTS CAP do formulário foi mapeado como valor terminal da tag <***Component Cap Name*** >, de maneira que a forma final dessa *tag* é: <***Component Cap Name*** > ::= { '*camera'* };

Os elementos ***captureImage*** e ***processImage*** listados no campo BEHAVIOR CAP do formulário foram mapeados como valores terminais da *tag* <***Behavior Cap***> da sintaxe, de maneira que a forma final dessa *tag* é: <***Behavior Cap***> { '*captureImage'* | '*processImage'* };

Os elementos **100**, **70** e **60** listados no campo PARAMETERS do formulário foram mapeados como valores terminais da *tag* < ***Parameter*** > da sintaxe, de maneira que a forma final dessa *tag* é: < ***Parameter*** > ::= { ‘100’ | ‘70’ | ‘60’ };

Os elementos ***On***, ***Off***, ***Auto***, **100**%, **70**% e **60**% listados nos campos STATES IN e STATES OUT do formulário foram mapeados como valores terminais da tag <***State***> da sintaxe, de maneira que a forma final dessa *tag* é: <***State***> ::= { ‘*On’* | ‘*Off’* | ‘*Auto’* | ‘100%’ | ‘70%’ | ‘60%’ };

Os elementos ***Switch*** e ***Sensor*** listados no campo CLASSES IN do formulário foram mapeados como valores terminais da *tag* <**Component In Class**> da sintaxe, de maneira que a forma final dessa *tag* é: <**Component In Class**> ::= { ‘*Switch’* | ‘*Sensor’* };

O elemento ***Light*** listado no campo CLASSES OUT do formulário foI mapeado como valor terminaL da *tag* <**Component Out Class**> da sintaxe, de maneira que a forma final dessa *tag* é: <**Component Out Class**> ::= { ‘*Light’* };

O elemento ***Camera*** listado no campo CLASSES CAP do formulário foI mapeado como valor terminal da *tag* <**Component Cap Class**> da sintaxe, de maneira que a forma final dessa *tag* é: <**Component Cap Class**> ::= { ‘*Camera’* };

Por fim, o nome do caso de testes foi utilizado para mapear um valor terminal da *tag* < *Test Case Class* >da sintaxe, de maneira que a forma final dessa *tag* é: < ***Test Case Name*** > ::= { *TestCaseAutomaticLight* }.

A seguir é apresentada a sintaxe completa com todos os seus elementos terminais definidos.

<Test Suit> ::= ***<Test Suit Name>*** **‘ :** TestSuit’ ‘ { ’ (***<Test Case>*** ‘;’) * ‘ } ;’

<Test Case> ::= ***<Test Case Name>*** **‘:’** ***<Test Case Class>*** ‘{ ’***<Test Setup>*** ‘;’ (***<Test Step>*** ‘;’) * ‘ };’

<Test Setup> ::= ‘ Setup: { ’ (‘(‘ (***<Component In Name> <Component In Class>*** [‘ : state = ’***<State>***]) * ‘)’ | (‘ (’***<Component Out Name> <Component Out Class>*** [‘ : state = ’***<State>***]) * ‘)’| ‘ (’(***<Component Cap Name> <Component Cap Class>*** [‘ : state = ’***<State>***] ‘)’) * ‘ };’

<Test Step> ::= ***<Step Name>*** ‘: {’ [(‘(’ ***<Step In>*** ’)’) *] [(‘ : (’ ***<Step Out>*** ‘)’) *] [(‘ : (’ ***<Step Cap>*** ‘)’)*] ‘: (’ ***<Assert>*** ’) ;’

<Step In> ::= ***<Behavior In>*** **‘(‘**[***< Parameter >*** *] **’)’** ‘ , ’ ***<Component In Name >*** ‘;’

<***Step Cap***> ::= <***Behavior Cap***> ‘(’[(< ***Parameter*** >) *] ’)’ ‘ , ’ <***Component Cap Name***> ‘;’

<***Step Out***> ::= <***Behavior Out***> **‘(’** [(< ***Parameter*** >) *] ’)’ ‘ , ’ <***Component Out*** **Name**> [‘ : ’ <***State***>]‘;’

<***Assert***> ::= ‘{ assert({ ’ <***Component Out Name***> ‘ }, {’ <***Component Cap Name***> ‘ }), ’ <***Test Case Name***> ‘ };’

<***Behavior In***> ::= { ‘switchOn’ | ‘switchOff’ | ‘switchAuto’ | ‘setLuminosity’ }

<***Component In Name***> ::= { ‘lightSwitch’ | ‘luminositySensor’ }

<***Behavior Cap***> { ‘captureImage’ | ‘processImage’ }

<***Component Cap Name*** > ::= { ‘camera’ }

<***Behavior Out***> ::= { ‘turnOn’ | ‘turnOff’ }

<***Component Out Name***> ::= { ‘headLight’ }

<***Parameter*** > ::= { ‘100’ | ‘70’ | ‘60’ }

<***State***> ::= { ‘On’ | ‘Off’ | ‘Auto’ | ‘100%’ | ‘70%’ | ‘60%’ }

<***Component In Class***> ::= { ‘Switch’ | ‘Sensor’ }.

<***Component Out Class***> ::= { ‘Light’ }.

<***Component Cap Class***> ::={ ‘Camera’ }.

<***Test Case Class***> ::= { ‘TestCaseAutomaticLight’ }.

<***Test Suit Name***> ::= < ***Character*** > (<***Character***> | <***Number***>)*

<***Test Case Name***> ::= <***Character***> (<***Character***> | <***Number***>)*

<***Step Name***> ::= <***Character***> (<***Character***> | <***Number***>)*

<***Character***> ::= { a, b, c, d, e, f, g, h, i, j, k, l ,m, n, o, p, q, r, s, t, u, v, w, y, z, A, B, C, D, E, F, G, H, I, J, K, L ,M, N, O, P, Q, R, S, T, U, V, W, Y, Z}

<***Number***> ::= { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

5.2.1.4 Transcrição dos casos de teste

Seguindo o fluxo de atividades proposto na Figura 11, o caso de teste em estudo foi transcrito utilizando-se a sintaxe definida nas atividades anteriores. Em um primeiro momento foi criada a sentença que define o setup inicial do teste, onde foram listadas as instâncias de classes necessárias para a realização do teste em questão.

Setup: { (lightSwitch Switch: state = OFF) (sensor LuminositySensor: state = 0) (headLight Light: state = OFF) (camera Camera) };

Em seguida, tomando como base a sequência de seis passos descritos no caso de teste, foi feita a transcrição dessas atividades para passos escritos utilizando-se a sintaxe definida para este caso.

1. Colocar a **chave** para posição desligada, e o **farol** também deve estar desligado;

Passo1: { switchOff(), lightSwitch } : { turnOff(), headlight } : { captureImage(), camera } : { processImage(), camera } : { assert({ getState(), headlight }, { getState(), camera }), testCase1 };

2. Mudar a posição da **chave** para ligada e o **farol** deve acender;

Passo2: { switchOn(), lightSwitch } : { turnOn(), headlight } : { captureImage(), camera } : { processImage(), camera } : { assert({ getState(), headlight }, { getState(), camera }), testCase1 };

3. Retornar a **chave** para posição desligada e o **farol** deve apagar;

Passo3: { switchOff(), lightSwitch } : { turnOff(), headlight } : { captureImage() , camera } : { processImage(), camera } : { assert({ getState(), headlight }, { getState(), camera }), testCase1 };

4. Mudar a **chave** para posição automático e o **farol** deve permanecer apagado;

Passo4: { switchAuto(), lightSwitch } : { turnoff(), headlight } : { captureImage() , camera } : { assert({ getState(), headlight }, { getState(), camera }), testCase1 };

5. Mudar o valor do **sensor** de luminosidade para 70% e o **farol** deve permanecer apagado;

Passo5: { setLuminosity(70), luminositySensor } : { turnOff, headlight } : { captureImage() , camera } : { assert({ getState(), headlight }, { getState(), camera }), testCase1 };

6. Mudar o valor do **sensor** de luminosidade para 60% e o **farol** deve acender.

Passo6: { setLuminosity(60), luminositySensor } : { turnOn(), Headlight } : { captureImage() , camera } : { assert({ getState(), headlight }, { getState(), camera }), testCase1 };

O resultado final dessa transcrição é mostrado abaixo. Esse trecho representa as ações e estados esperados para cada passo listado no caso de teste utilizando-se as regras de sintaxe definidas neste estudo. A partir dessa transcrição, todos os casos de teste, escritos utilizando-se qualquer outra forma ou sintaxe podem ser transcritos de uma maneira única e padronizada para esta proposta.

```
testSuit1 {
  testCase1 : TestCase1{
      Setup:  { ( lightSwitch Switch: state = OFF ) ( sensor LuminositySensor: state = 0 )
          ( headLight Light: state = OFF ) ( camera Camera ) };
            Passo1: { switchOn(), lightSwitch } : { turnOn(), headlight } : {
              captureImage(), camera } : { processImage(), camera } : { assert( {
              getState(), headlight }, { getState(), camera } ), testCase1 };
            Passo2: { switchOn(), lightSwitch } : { turnOn(), headlight } : {
              captureImage(), camera } : { processImage(), camera } : { assert( {
              getState(), headlight }, { getState(), camera } ), testCase1 };
            Passo3: { switchOff(), lightSwitch } : { turnoff(), headlight } : {
              captureImage() , camera } : { processImage(), camera } : { assert( {
              getState(), headlight }, { getState(), camera } ), testCase1 } ;
            Passo4: { switchAuto(), lightSwitch } : { turnoff(), headlight } : {
              captureImage() , camera } : { assert( { getState(), headlight }, { getState(),
              camera } ), testCase1 };
            Passo5: { setLuminosity(70), luminositySensor } : { turnOff, headlight } : {
              captureImage() , camera } : { assert( { getState(), headlight }, { getState(),
              camera } ), testCase1 };
            Passo6: { setLuminosity(60), luminositySensor } : { turnoff(), Headlight } : {
              captureImage() , camera } : { assert( { getState(), headlight }, { getState(),
              camera } ), testCase1 };
   };
};
```

5.3 Modelo Independente de Plataforma

5.3.1 Criação do diagrama de classes

De posse do formulário de dados provindo da atividade de análise do documento de testes, pode-se iniciar a fase de modelagem da parte estrutural da ferramenta de execução automatizada de testes, a partir da criação do diagrama de classes da mesma. As próximas subseções apresentam as classes criadas a partir dos dados organizados e transcritos até o momento.

5.3.1.1 Definição das classes ComponentOut

Seguindo as diretrizes estabelecidas na seção 4.2.1.1, foram criadas classes responsáveis por modelar elementos constituintes do SUT e seus comportamentos. Partindo da estrutura básica definida na seção 4.2.1 e apresentadas na Figura 13, foram criadas a classe filha de *ComponentOut* nomeada *Light* e a classe do tipo *Enumeration LightStates*, respectivamente representadas em destaque nas cores rosa e verde na Figura 19.

Seguindo as diretrizes para a modelagem das classes estabelecidas na seção 4.2.1.1, os dados do campo BEHAVIOR OUT do formulário foram cada um mapeados na forma de um método dessa classe, assim como os valores definidos para a classe *LightStates* que possui o esterótipo *Enummeration* foram baseados nos dados descritos no campo STATES OUT do mesmo formulário.

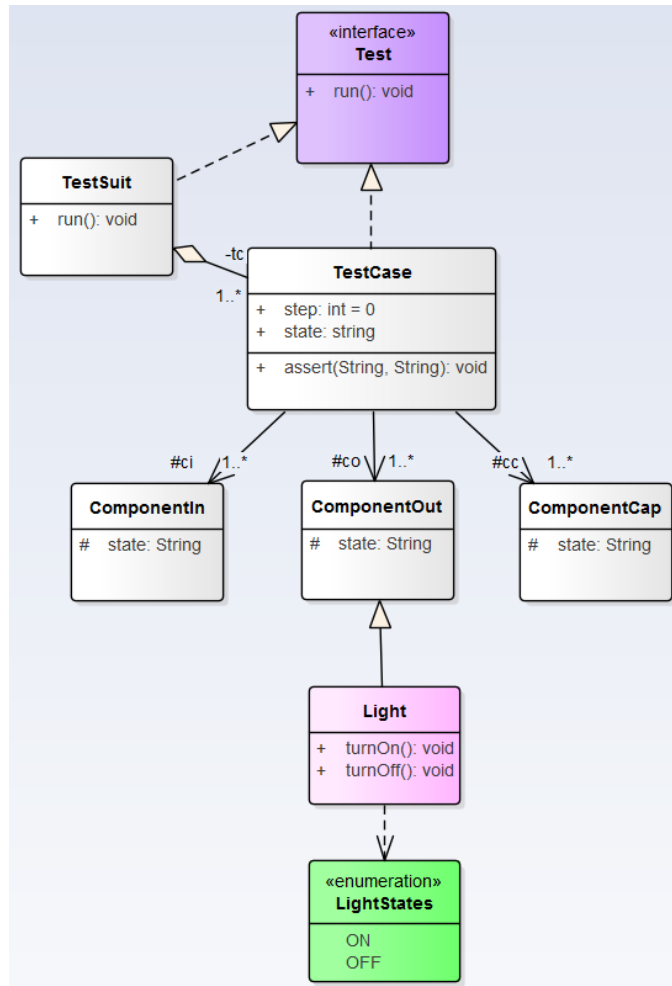

Figura 19: Acréscimo da classe Light por meio de especialização da classe ComponentOut.

5.3.1.2 Definição das classes ComponentIn

A etapa seguinte foi a de criação das classes de componentes responsáveis por modelar as entradas ou estímulos fornecidos a um sistema embarcado. Seguindo a mesma metodologia estabelecida na seção anterior, foram modeladas as classes *Switch*, que representa uma chave de 3 posições e uma classe *LuminositySensor*, que representa um sensor de luminosidade. A Figura 20 apresenta a modelagem dessas classes.

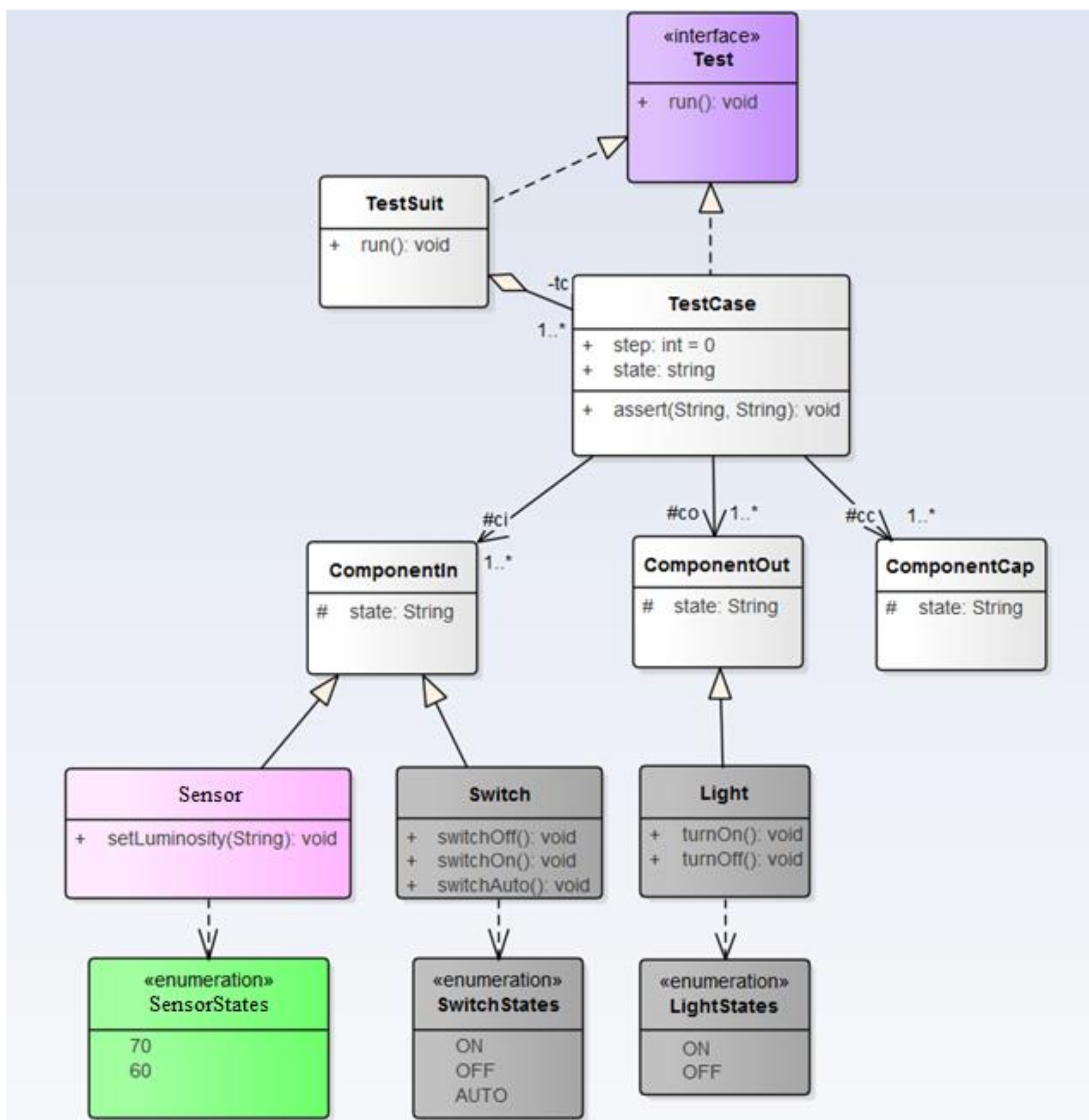

Figura 20: Acréscimo das classes LuminositySensor e Switch por meio de especialização da classe ComponentIn.

5.3.1.3 Definição das classes ComponentCap

Seguindo as diretrizes propostas na seção 4.2.1.3 foram criadas classes responsáveis por modelar componentens de apoio ao teste. Para o caso em estudo, foi estabelecida a necessidade de uma câmera reponsável por capturar e processar imagens do SUT por meio de técnicas de visão computacional. Essa escolha se deu devido à característica visual do comportamento do SUT.

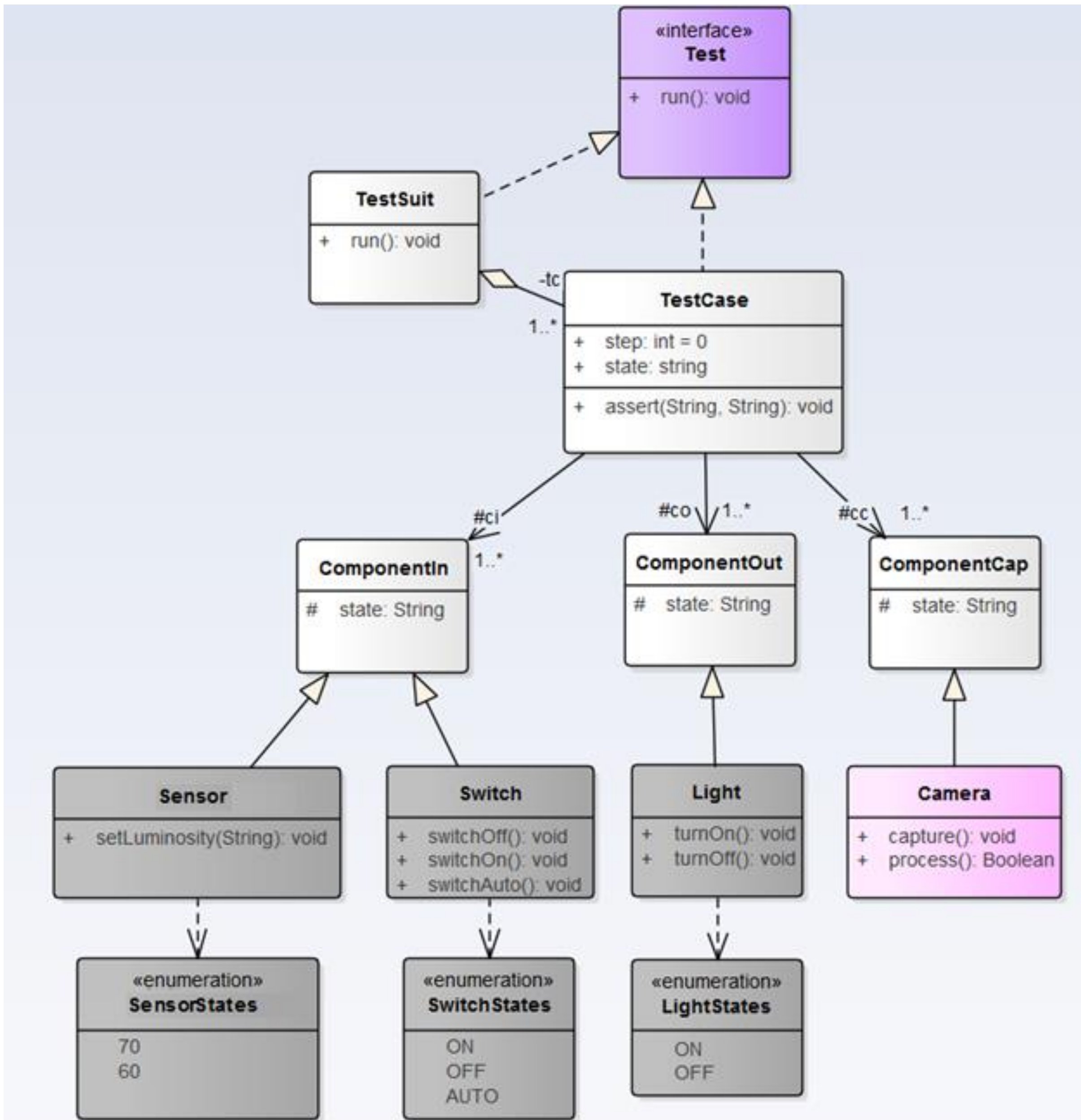

Figura 21: Acréscimo da classe Camera por meio de especialização da classe ComponentCap.

5.3.1.4 Definição das classes TestCase

Por fim, e seguindo o que foi proposto na seção 4.2.1.4, foi inserida ao diagrama de classes a classe *TestCaseAutomaticLight*, a qual é responsável pela execução do caso de teste em estudo.

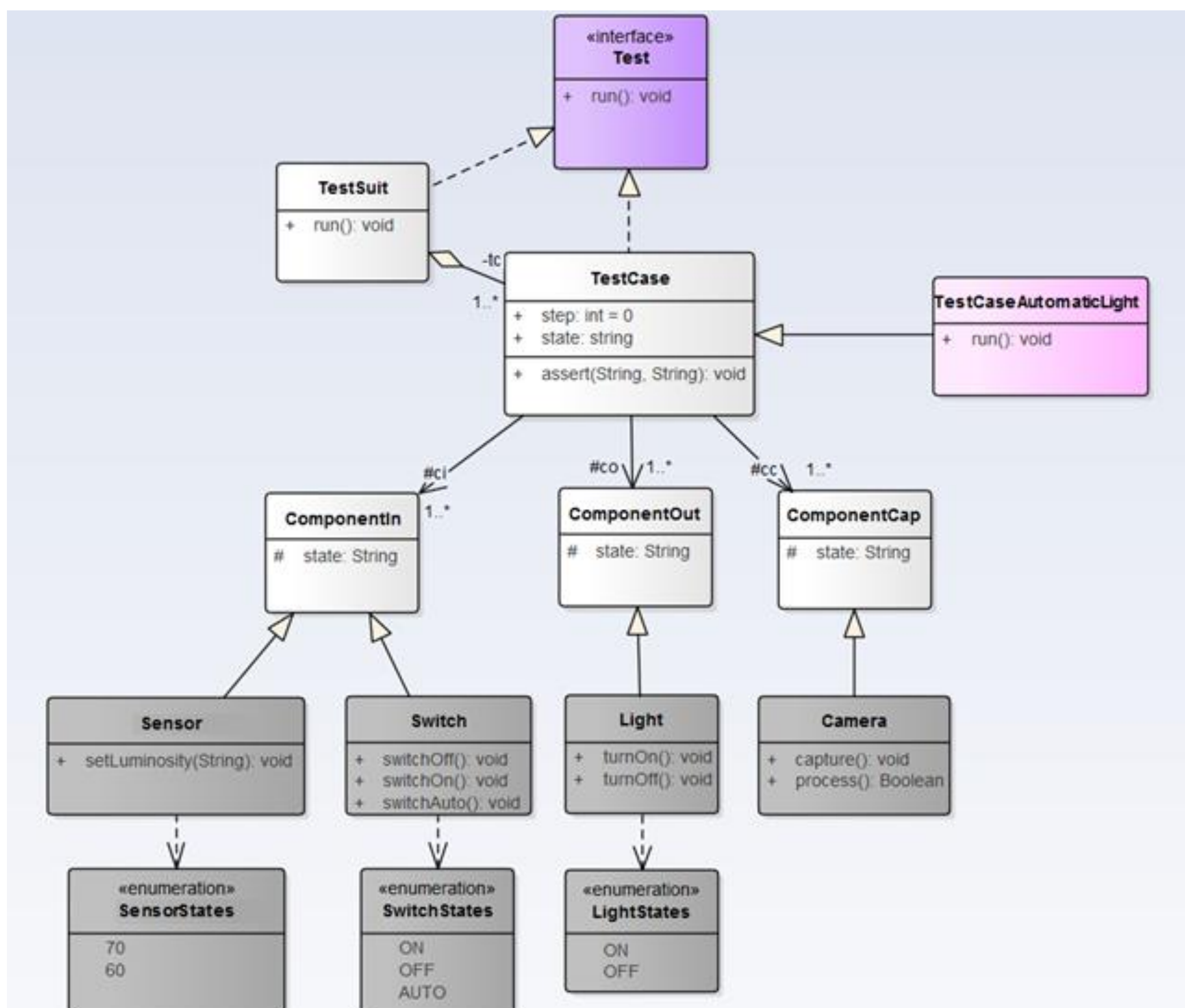

Figura 22: Modelo PIM completo do diagrama de classes do sistema.

Uma vez finalizado esse processo, obteve-se a estrutura de classes completa necessária para a descrição estrutural do sistema, restando as implementações comportamentais dos métodos declarados em cada classe, processo este que será descrito na seções subsequentes.

5.3.2 Diagramas de sequência

A modelagem comportamental da ferramenta de automação de testes foi realizada por meio do diagrama de sequência da UML. A partir da descrição do caso de teste escrita utilizando-se a metodologia e sintaxe definida nesta proposta, foi feita a transformação do mesmo em um diagrama de sequência que modela o comportamento do método *run*() da classe *TestCaseAutomaticLight*.

Aplicando-se as diretrizes propostas na seção 4.2.2.1 à descrição do caso de teste em estudo, foi gerado o diagrama de sequência correspondente ao mesmo. Nesta seção é apresentada na Figura 23 uma parte do diagrama de sequência, uma vez que o diagrama em sua totalidade seria de difícil vizualização dada a sua grande extensão.

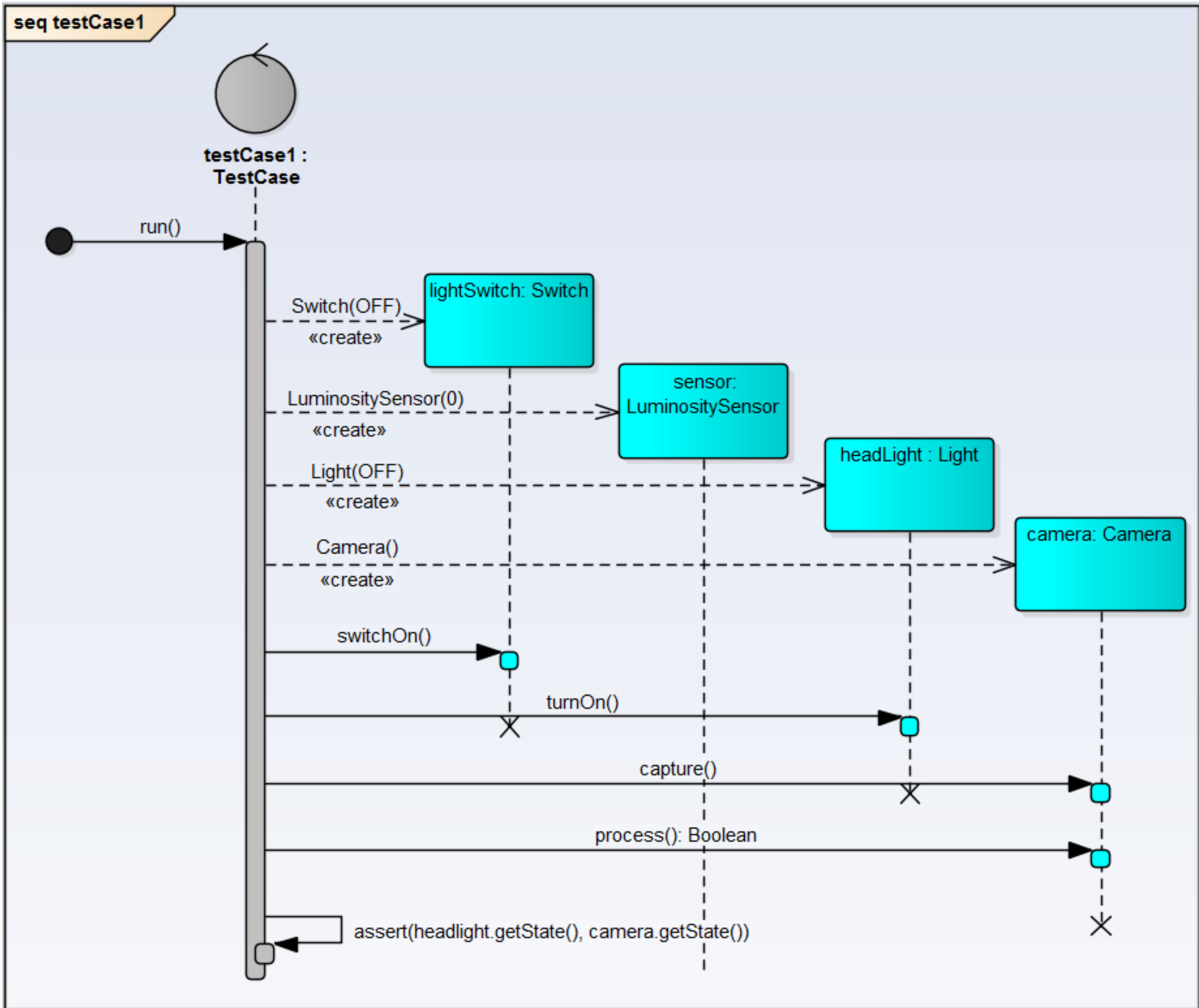

Figura 23: Diagrama de sequência referente aos primeiros passos do caso de teste.

O modelo apresentado na Figura 23 corresponde a um diagram de sequência criado a partir das linhas de *setup* e do primeiro passo do caso de teste.

Setup: { (lightSwitch Switch: state = OFF) (sensor LuminositySensor: state = 0) (headLight Light: state = OFF) (camera Camera) };

Passo1: { switchOn(), lightSwitch } : { turnOn(), headlight } : { captureImage(), camera } : { processImage(), camera } : { assert({

getState(), headlight }, { getState(), camera }), testCase1 };

As informações presentes na linha de *Setup* foram mapeadas na forma de instanciações de objetos a partir da criação de *Lifelines*. Cada objeto listado dentro da linha de Setup possui um nome, a classe da qual é instância além de parâmetros a serem passados ao método contrutor da mesma. Dessa forma, cada element dessa linha resulta na criação de uma *Lifeline* nomeada como “nome da instância”: “Nome da classe” e é elemento alvo de uma mensagem com o estereótipo “<<*Create*>>” com o nome do constructor dessa classe. O resultado dessa transformação apresenta-se destacado na cor laranja na Figura 24.

A linha que descreve o passo 1 por sua vez, apresenta uma sequência de invocações de métodos, onde cada invocação apresenta o nome do método a ser invocado, o objeto ao qual esse método pertence e os parametros necessários a esse metodo. Dessa forma, cada invocação de método listado no passo 1 resultou na criação de uma mensagem apontando para o respectivo objeto alvo. O resultado desta trasnformação apresenta-se destacado na cor azul na Figura 24.

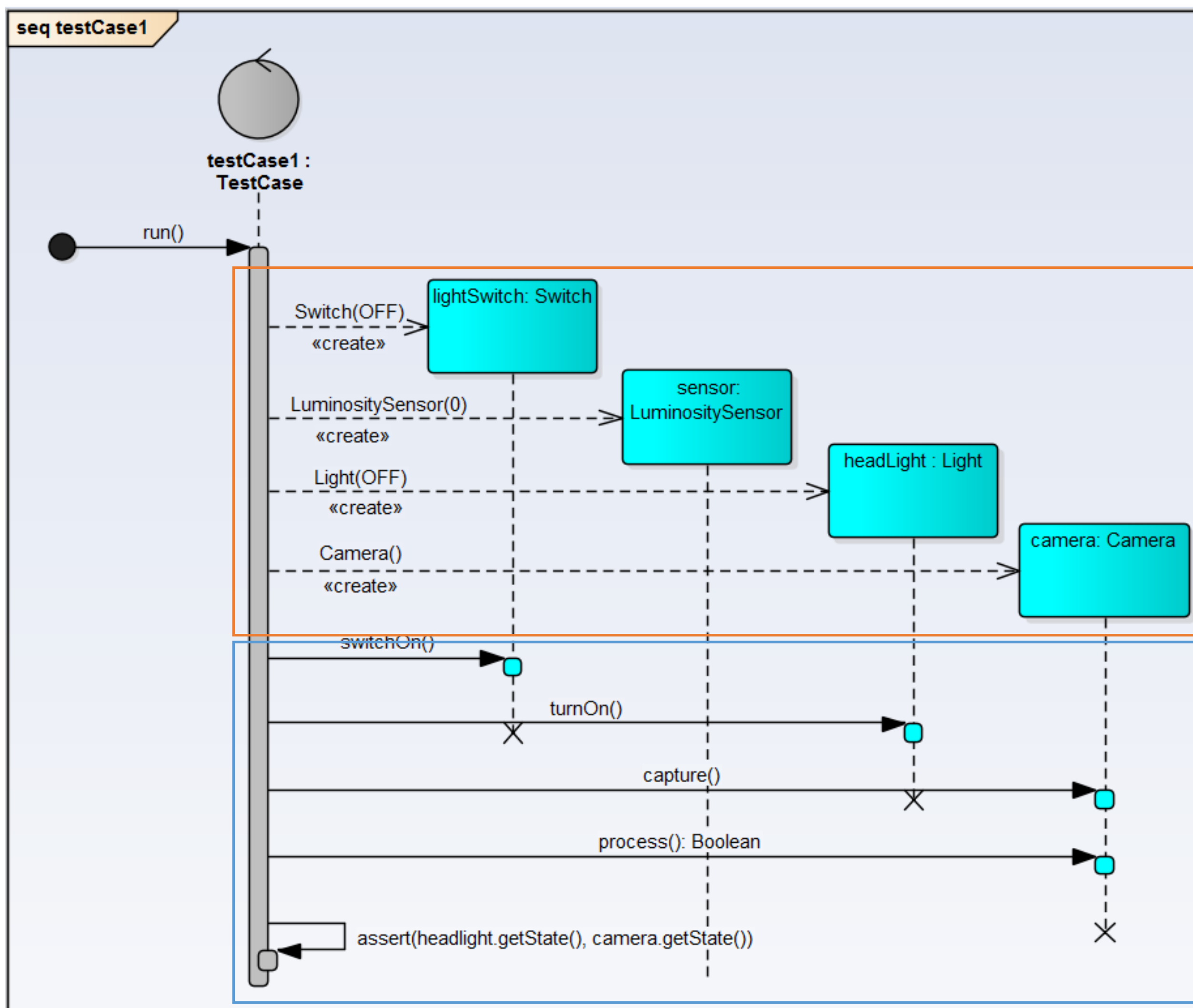

Figura 24: Diagrama de sequência diferenciando as ações de *setup* e do primeiro passo do caso de teste.

5.4 Modelo Específico de Plataforma – PSM

5.4.1 Aplicação de transformações MDA automatizadas nos diagramas de classe e sequência

Os modelos gerados e descritos até o momento são modelos independentes de plataforma escritos utilizando-se a linguagem UML. A partir desses modelos – PIM, a proposta do MDA é a de aplicar transformações automáticas para modelos específicos de plataforma – PSM. A ferramenta EA utilizada para a modelagem de diagramas neste trabalho dá suporte à transformações automáticas de modelos estruturais como o diagrama de classes.

No entanto essa ferramenta ainda não possui suporte a transformações que envolvem modelos comportamentais, como o diagrama de sequência. Apesar desse fato, as transformações de modelos comportamentais podem ser viabilizadas a partir da estrutura XMI do diagrama, na qual podem ser encontradas todas as informações necessárias para o mapeamento de elementos.

No entanto, uma vez que dentro da proposta deste estudo os modelos comportamentais apresentam características e escopo bem definido e limitado à aplicações voltadas à execução de testes funcionais, os diagramas de sequência em sua forma PIM são exatamente identicos aos diagramas PSM. Assim, as transformações de PIM para PSM abordadas neste estudo são exclusivamente de modelos estruturais, no caso os diagramas de classe.

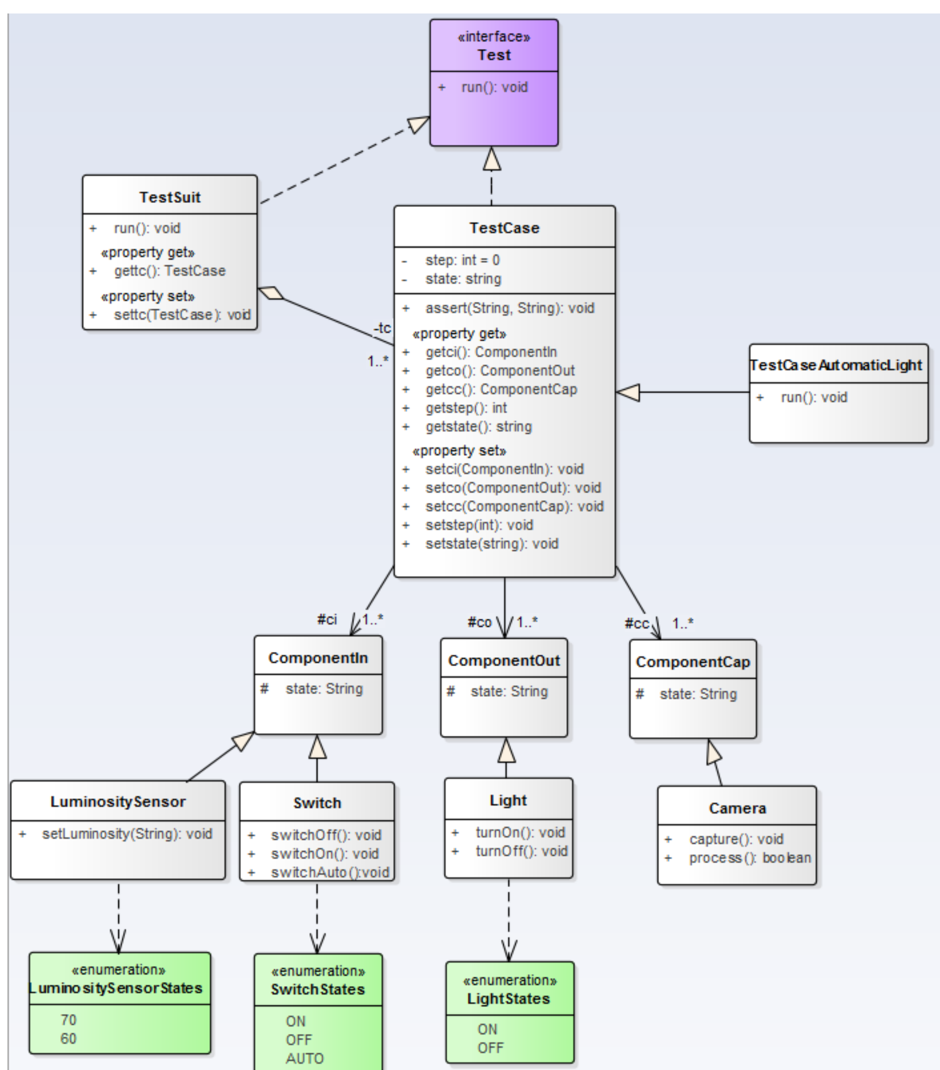

Figura 25: Modelo PSM do diagrama de classes.

Como pode ser visto na Figura 25 o modelo resultante desta transformação possui algumas diferenças quando comparado ao seu modelo de origem (Figura 22), como a definição e encapsulamento de atributos privados e acréscimo dos métodos *set's* and *get's* para as associações entre classes

5.5 Código

5.5.1 Transformação de PSM para Código

A transformação do modelo PSM para código do sistema pode ser realizada por meio da aplicação de transformações já definidas na ferramenta EA. No entanto, conforme discutido na seção 4.4.1, as trasnformações disponibilizadas pela ferramenta são essencialmente transformações de diagramas estruturais como as apresentadas na Figura 26, sendo que a ferramenta, até o presente momento não possui suporte para transformações de PSM para código de modelos comportamentais.

```
1 package Cluster;
2
3
4 public class TestSuit implements Test {
5
6     private TestCase tc;
7
8     public TestSuit(){
9
10    }
11
12    public void finalize() throws Throwable {
13
14    }
15
16    public TestCase gettc(){
17        return tc;
18    }
19
20    public void run(){
21
22    }
23
24    /**
25     *
26     * @param newVal
27     */
28    public void settc(TestCase newVal){
29        tc = newVal;
30    }
31
32 }
```

```
1 package Cluster;
2
3
4 public class Camera extends ComponentCap {
5
6     public Camera(){
7
8     }
9
10    public void finalize() throws Throwable {
11        super.finalize();
12    }
13
14    public void capture(){
15
16    }
17
18    public boolean process(){
19        return false;
20    }
21
22 }
```

Figura 26: Código estrutural gerado automaticamente pela ferramenta EA por meio de transformações MDA.

Todavia, uma vez que a ferramenta EA suporta o padrão XMI, é possível

realizar o mapeamento completo dos elementos de um modelo para seus respectivos elementos de código, viabilizando-se assim a transformação automática desses modelos em código do sistema.

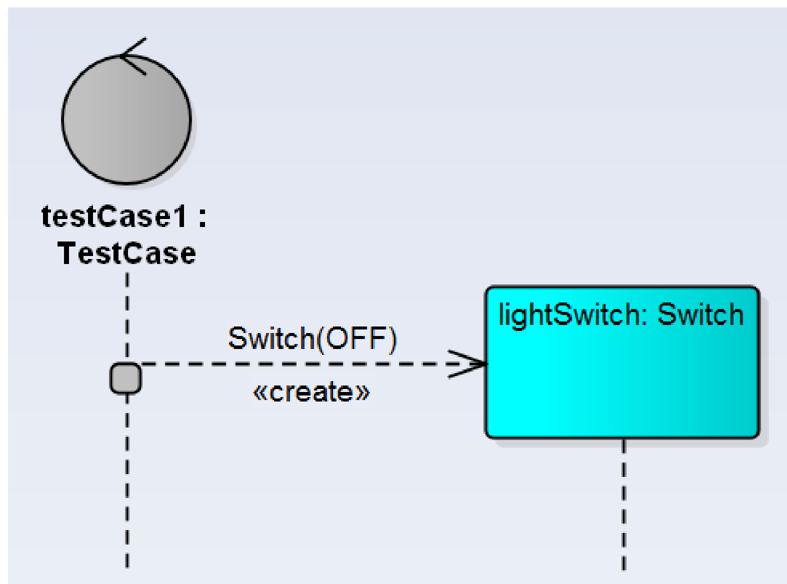

Figura 27: Trecho do diagrama de sequência submetido a mapeamento de elementos via documente XMI.

Como meio de exemplificar a realização desta trasformação é apresentado o mapeamento para o código de um trecho do diagrama de sequencia, mostrado na Figura 27 a partir da versão em XMI do mesmo. Alguns elementos e atributos específicos e de relevância para esta conversão são listados abaixo:

- <**UML**:**ClassifierRole**>: Um ClassifierRole é um elemento utilizado para indicar a presença de uma Lifeline no diagrama, sendo o atributo "***name***" utilizado para armazenar o nome da instancia a qual se refere a *Lifeline*.
- <**UML:Message** name="nome do metodo"/>: O elemento Message é utilizado para indicar a presença de uma mensagem entre dois elementos no modelo. Esse elemento possui entre outros, o atributo "***name***" que contém o nome do método invocado e os atributos "***source***" e "**target**" que especificam os elementos fonte e alvo dessa mensagem.
- <**UML:TaggedValue**>: O elemento TaggedValue é utilizado para armazenar valores rotulados que servem como meios de descrição de características de um ClassifierRole, de uma mensagem ou outros elementos. Possui entre outros, um atributo nomeado "***tag*"** com o valor "***classname***", e um atributo "***value***" que recebe o

nome da classe cujo o objeto é instância. Alternativamente, o atributo “***tag***” pode receber outros valores como “***stereotype***" que determina o tipo de estereótipo desse elemento, “**mt**” que determina o nome de um função ou método e seus parâmetros entre parênteses caso existam

Utilizando os elementos citados acima foi possível realizar o mapeamento e transformação do diagrama de sequência Figura 11, sendo o resultado desta transformação a linha de código apresentada na Figura 28.

A linha de código da Figura 28 apresenta a sintaxe para a instanciação de um objeto na linguagem Java. Para a geração automática deste trecho de código a partir do diagrama de sequência da Figura 27 foram utilizados os rótulos e atributos presentes no XMI.

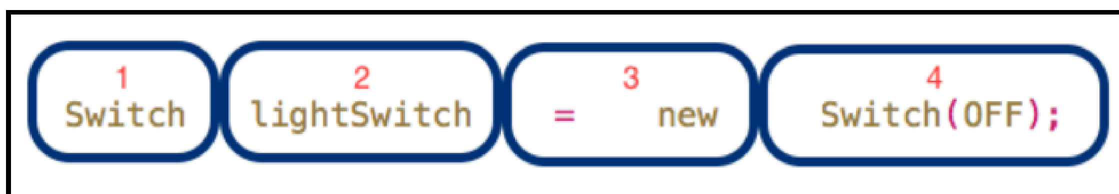

Figura 28: Linha de código em linguagem Java referente a instânciação de um objeto.

Na Figura 28 o código foi divido em quatro partes enumeradas onde:

1. O primeiro trecho corresponde ao tipo da classe a ser instanciada, e é encontrado no documento XMI do modelo no rótulo **<UML:TaggedValue tag=“classname” value=“Switch”/>** onde o valor do atributo “***value”*** corresponde ao nome da classe.
2. Este trecho corresponde ao nome do objeto instanciado. Este dado pode ser encontrado no documento XMI do modelo no rótulo **<UML:ClassifierRole name=“lightSwitch”/>**, onde o valor do atributo “***name***” corresponde ao nome do objeto a ser instanciado;
3. Este trecho corresponde ao comando reservado pela linguagem Java “***new*** “ que tem como função alocar espaço em memória para uma nova instancia de uma classe. Este dado pode ser interpretado a partir do XMI dentro do escopo do rótulo **<UML:Message>** a partir da tag **<UML:TaggedValue tag=“stereotype” value=“create”/>**. O estereotipo “***create***" é utilizado para indicar a criação de novos objetos, de forma que a existência deste estereótipo pode ser transcrita em Java com o uso do comando “***new***”;
4. Este trecho de código corresponde a invocação do método construtor da classe

instanciada e seus argumentos caso existam. Esta informação pode ser encontrada dentro do escopo do rótulo <**UML:Message**> a partir da tag <**UML:TaggedValue tag="mt" value="Switch(OFF)"**/>.

Ressalta-se ainda que o padrão XMI estipula as regras para a criação de cada uma destas tags baseadas na UML, de forma que todas os elementos presentes na definição da UML proposta pela OMG podem ser mapeados em respectivas *tags* XMI. [71].

O procedimento acima descrito apenas exemplifica o processo de mapeamento e transformação do diagrama de sequência para o código do sistema, sendo que o mesmo processo pode ser realizado para todos os demais elementos presentes no diagrama.

DESENVOLVIMENTO MANUAL DO CODIGO

Por fim, após a geração do código estrutural do sistema e também do código comportamental da classe *TestCaseAutomaticLight* através da aplicação das transformações MDA, foi realizada a implementação comportamental das classes *LuminositySensor*, *Light*, *Switch* e *Camera*. Os métodos presentes em cada uma dessas classes foram implementados, sendo de maneira geral, métodos responsáveis pelo envio de sinais binários a partir da porta paralela do computador para o SUT ou algoritmos de processamento digital de imagens como filtros de média e binarização no caso dos métodos da classe *Camera*.

Como meio de avaliação da condição acesa ou apagada da espia de farol, foi utilizado um algoritmo de contagem de pixels brancos na imagem capturada e processada, sendo que caso o número de *pixels* brancos represente uma porcentagem maior que um limite determinado, a espia é considerada acesa e esse resultado foi utilizado para a avaliação do teste.

5.5.2 Execução

Após a complementação e finalização das implementações comportamentais restantes, o programa foi compilado e executado em um computador pessoal ao qual foram ligadas a camera de captura e o cluster automotivo conforme a organização mostrada na Figura 29.

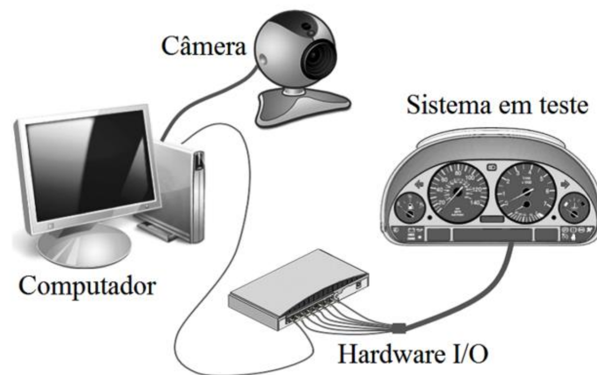

Figura 29: Organização física do sistema de automação de testes funcionais para um *cluster* automotivo.

O programa ativou com sucesso a luz indicativa de farol aceso assim como realizou a captura e processamento da imagem. Esses passos podem ser observados nas Figuras 30 e 31 respectivamente.

Figura 30: Espia indicativa de farol acionada via entradas do SUT.

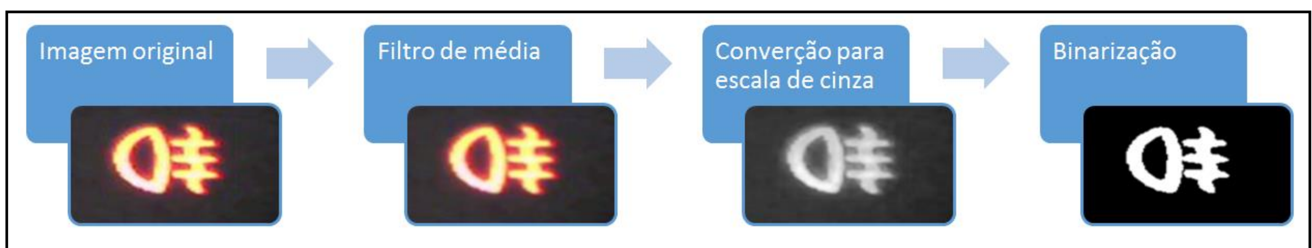

Figura 31: Processamento digital de imagem aplicado à imagem capturada pelo sistema por meio da camera digital.

5.6 Considerações do Capítulo

Neste capítulo foi apresentada uma experimentação prática da proposta de processo de desenvolvimento de sistemas de automação de testes funcionais para sistemas embarcados apoiado pelas diretrizes do MDA apresentada no capítulo quatro desse trabalho.

A criação do formulário e da sintaxe em BNF demonstrou a viabilidade de conversão de CIM para PIM para um caso de teste específico, a partir do qual foi possível fazer um mapeamento de elementos para transformação dos modelos CIM, para modelos em diagramas de classe e sequência. Dessa forma, é possível que, a partir do modelo CIM sejam executadas transformações automática a partir de um mapeamento dos elementos do BNF da sintaxe definida neste estudo para as *tags* de um documento XMI de diagramas de classes/sequência, bastando para isso que uma ferramenta de transformação que obedeça às regras, tanto do XMI quanto da sintaxe definida neste estudo seja implementada.

Observou-se ainda que a ferramenta adotada para este estudo possui um bom suporte para transformações estruturais entre modelos, principalmente com relação ao diagrama de classes da UML, possuindo inclusive mapeamentos já prontos para diversas plataformas. Porém, a mesma ferramenta não apresenta suporte no que tange a transformações de diagramas comportamentais como o diagrama de sequência. Entretanto, vale ressaltar que esta característica não limita-se somente à ferramenta adotada neste estudo. De maneira geral, ferramentas de apoio ao MDA normalmente suportam somente transformações automáticas da parte estrutural e não comportamental do sistema em desenvolvimento. Além disso, as transformações automáticas entre modelos geralmente são realizadas partir de modelos PIM em diante, não contemplando métodos de transformação de CIM para PIM [46].

Apesar desse fato, verificou-se a viabilidade da construção de uma ferramenta de transformações de modelo comportamentais uma vez que a estrutura XMI disponibiliza meios para identificação e extração das informações necessárias para as respectivas transformações, conforme demonstrado no exemplo.

Além disso, é comum que um diagrama de sequência nem sempre contemple todo o conteúdo do código final de um produto de software. No entanto, neste trabalho buscou-se criar uma estrutura e organização do processo de forma que fosse possível construir um diagrama de sequência de um caso de teste completo, cujo código final estivesse totalmente contemplado no mesmo, viabilizando assim uma conversão automática também da parte comportamental dos casos de teste. Apesar disso, destaca-se ainda que esta proposta

ainda não possibilita que 100% do código final do sistema de automação de testes seja gerado automaticamente, pois ainda foi necessária a implementação manual dos métodos *setLuminosity*, *SwitchOn*, *switchOff*, *switchAuto*, *turnOn*, *turnOff*, *capture* e *process* presentes nas classes *LuminositySensor*, *Switch*, *Light* e *Camera* do modelo estrutural apresentado na Figura 22. No entanto, Hoffman [32] ressalta que na indústria de software embarcado, é comum que testes sejam realizados inúmeras vezes ao longo do ciclo produtivo, ou que testes similares sejam desenvolvidos e aplicados a produtos com características semelhantes ou ainda que sejam feitas revisões em testes já existentes frente a atualizações ou novas versões de um produto. Nesse sentido, os códigos criados manualmente podem ser reutilizados em sistemas de automação de outros produtos. As implementações manuais de código dentro desta proposta são restritos a comportamentos de componentes responsáveis pela interação do sistema de automação com o SUT, sendo que as sequências e lógicas envolvidas na realização de um teste são automaticamente transformadas em código via MDA. Analisando-se essa característica, diferentes casos de teste, mas que utilizam os mesmo meios de interação com o SUT podem ser modelados, fazendo o reuso desses componentes.

6 CONCLUSÃO

Neste trabalho foi apresentada uma proposta apoiada pelo padrão de desenvolvimento MDA definido pela OMG. O estudo e compreensão do MDA contribuíram para o desenvolvimento de um processo para criação de ferramentas para um domínio específico de sistemas de automação de teste para sistemas embarcados.

O processo abrangeu as três fases de transformações de modelos propostos para o MDA, sendo elas as transformações de CIM para PIM, PIM para PSM e PSM para código para o referido domínio.

As transformação de CIM para PIM normalmente são pouco abordadas nos trabalhos, visto que a própria OMG não estabelece um padrão para o CIM, portanto não tem uma padronização dessas transformações. Diante disso foram idealizadas uma sintaxe utilizando BNF para normatização de documentos de teste, apoiada ainda pela estruturação de um formulário de dados responsável por condensar as principais informações a respeito dos testes a serem automatizados, proporcionando assim, um padrão de escrita para os documentos CIM utilizados nesta proposta. Dessa forma, foi possível fazer um mapeamento de elementos da sintaxe para transformação do conteúdo descritivo dos testes para modelos PIM adotando os diagramas de classe e de sequência da UML.

As transformações de modelos PIM para PSM e de PSM para código também foram abordadas neste estudo, sendo que as transformações de modelos estruturais, definidos por diagramas de classes foram realizadas a partir do uso das implementações e mapeamentos já existentes em ferramentas CASE com suporte ao MDA existentes no mercado. Para as transformações de modelos comportamentais, no caso o diagrama de sequência, que normalmente as ferramentas CASE de MDA não possuem suporte, foi exemplificado um mapeamento a partir dos elementos de modelo descritos na estrutura XMI do mesmo mostrando assim a viabilidade desta transformação para o escopo de testes de software.

Com relação à transformação do diagrama de sequência em código, o resultado dessa transformação implica na implementação de alguns comandos dos métodos representados no modelo. Destaca-se principalmente que, no processo de transformação do diagrama de sequência para código, estruturado dentro do escopo deste trabalho, apenas o método *run*() (Figura 23) foi implementado de forma automática, visto que os demais comportamentos são apenas métodos receptores. Além disso, ressalta-se ainda o fato de que o

comportamento do método *run*() no diagrama descreve o caso de testes na sua completude, assim a transformação do mesmo resulta no código final sem complementações ou programações posteriores.

No entanto, destaca-se ainda que o código gerado seguindo as diretrizes do MDA ainda necessita de complementações codificadas manualmente referentes ao conteúdo dos métodos das classes que especializam as classes *CompoenentIn*, *ComponenteOut* e *ComponentCap*. Entretanto, esses métodos são de simples implementação, uma vez que não envolvem a lógica de funcionamento do SUT, o qual atua como uma caixa preta no processo. Esses métodos são geralmente responsáveis pelo envio de sinais de entrada para o SUT ou ainda pela captura de um comportamento esperado em resposta a essa entrada.

A implementação de algoritmos para interpretação das informações capturadas do SUT pode ser apontada como a etapa mais trabalhosa dessa fase. Porém, essas implementações realizadas manualmente no código podem ainda ser reaproveitados em diferentes casos de teste de uma mesma suíte, ou ainda ser reusados em testes futuros com características semelhantes.

Dessa forma, quando observado sob uma perspectiva de longo prazo, é factível que o uso do processo proposto neste trabalho tenha como produto, um repositório de modelos e códigos, que possibilitará a redução gradual da necessidade de codificação manual no processo, até o ponto onde sistemas de automação de testes possam ser gerados completamente a partir de modelos e com reuso de código de componentes. Essa característica, associada às vantagens inerentes ao MDA possibilita algumas vantagens para o processo de desenvolvimento e aplicação de testes automatizados para softwares embarcados uma vez que a modelagem realizada melhora significativamente a documentação dos testes, assim como faz com que o código gerado reflita fielmente o conteúdo dos modelos criados e possibilitando ainda a rastreabilidade de cada elemento e modelo utilizado no processo, facilitando assim a identificação dos mesmos sob uma ótica mais global do processo, tornando assim o processo de automação mais atrativo em relação à execução manual dos testes.

Além disso, foram apresentados outros trabalhos relacionados à mesma temática deste estudo, ou com propostas semelhantes [28] [48] [53] [54] [60]. Logo, em comparação com os trabalhos estudados e apresentados na revisão de literatura, é possível apontar alguns ganhos e perdas com relação à proposta deste estudo.

Percebeu-se que estes estudos de maneira geral são voltados a uma particularidade específica de um determinado ramo de atuação, como aplicações militares e de

entretenimento. Em contra ponto com este cenário, a proposta descrita neste estudo mostra-se mais abrangente, uma vez que o processo foi idealizado com o intuito de abranger os testes funcionais em sistemas embarcados diversos, empregando ainda uma experimentação voltada ao setor automotivo, normalmente bastante exigente em termos de teste de software. Dadas essas características, o processo pode ser considerando suficientemente genérico para ser aplicado em outros setores que fazem uso de sistemas embarcados. No entanto, essa tarefa será realizada em trabalhos ainda vindouros.

Quando comparado a trabalhos presentes na literatura que utilizam o MDA como meio norteador para a proposta, percebe-se que os modelos abordados são geralmente os modelos PIM e PSM, deixando uma lacuna no que tange à aplicação dos modelos CIM. Além disso, a aplicação de transformações automatizadas para modelos comportamentais, foram pouco abordadas nos trabalhos analisados. Em contraponto a esse cenário, este trabalho apresentou ganhos em ambos os quesitos, uma vez que foram estabelecidas formas para inserção dos modelos CIM ao processo de desenvolvimento para sistemas de automação de testes, abrangendo inclusive formas de automatização das transformações desses modelos para modelos PIM, assim como foram mapeadas formas para a realização de transformações automáticas de diagramas comportamentais para o domínio do problema.

Além disso, foram apontados na revisão bibliográfica, trabalhos que utilizavam ferramentas comercias para a implementação da automação de testes. Essas ferramentas possuem como principal atrativo, o fato de serem ferramentas já bastante difundidas dentro do meio industrial, assim como por apresentarem um elevado grau de maturidade e abrangência. No entanto, os custos associados para a aquisição dessas ferramentas são geralmente elevados, assim como o uso de linguagens de modelagem e programação proprietárias. Em contrapartida, a proposta descrita neste trabalho faz uso de padrões amplamente difundidos como a UML, promovendo assim o uso de uma linguagem de modelagem unificada, possibilitando assim, o uso de ferramentas diversas, desenvolvidas de acordo com os padrões estabelecidos pela OMG para o MDA.

Por fim, destaca-se ainda que as ferramentas com suporte ao MDA ainda encontram-se em um estado intermediário de maturidade, sendo que algumas limitações como a falta de suporte às transformações comportamentais podem representar um obstáculo à implementação dessa proposta.

Desta forma, o objetivo geral deste estudo visou a criação de um conjunto de modelos especializados para um domínio particular de aplicação: Sistemas de automação

de testes para dispositivos com sistemas embarcados. O trabalho alcançou este objetivo por meio de modelos desenvolvidos seguindo as diretrizes do MDA orientados pela arquitetura dos padrões UML e MOF, inclusive formalizando os processos de conversões de modelos CIM para PIM pois estabeleceu um processo que institui um formalismo que possa guiar o desenvolvimento de sistemas e arquiteturas de teste de software e que seja pautado por padrões amplamente difundidos e utilizados na área, como os adotados e mantidos pela OMG.

6.1 Trabalhos futuros

Como trabalhos futuros e complementares ao estudo realizado nesta proposta, sugerem-se um estudo mais aprofundado sobre as vantagens e desvantagens da integração das propostas de MDA com as propostas existentes para reuso de componentes e código. Propõe-se ainda um estudo mais aprofundado, comparativo e complementar sobre o processo aqui descrito com a proposta do MDT – *Model Driven Test*, que busca a geração automática de casos de teste derivados da modelagem do sistema.

Outro estudo futuro proposto, é o realizar a aplicação do processo definido neste trabalho sob a forma de uma experimentação metodológica dentro de um ambiente empresarial ou industrial do ramo de desenvolvimento de embarcados.

Além disso, objetiva-se ainda como um trabalho futuro, a extensão da sintaxe definida neste trabalho para abranger os demais elementos constituintes de um diagrama de sequencia, com especial ênfase para os fragmentos combinados *alt, opt, ref, loop* dentre outros, assim como avaliar a proposta após esses incrementos.

REFERÊNCIAS

[1] KARSAI, G. et al. *EVOLVING EMBEDDED SYSTEMS*. [S.l.]: Wiley, 2010.

[2] MALINOWSKI, A.; YU, H. *Comparison of Embedded System Design for Industrial Applications*. [S.l.]: IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, 2011.

[3] BARR, T. W.; SMITH, R.; RIXNER, S. *Design and Implementation of an Embedded Python Run-Time System*. [S.l.]: USENIX Annual Technical Conference, 2012.

[4] POLO, M. et al. *Test Automation*. [S.l.]: IEEE Software, v. 30, 2013.

[5] FEWSTER, M.; GRAHAM, D. *Software Test Automation*. 1ª. ed. Nova York: Addison-Wesley, 1999.

[6] BINDER, R. V. *Testing Object-oriented Systems:* Models, Patterns, and Tool. Boston: Addison-Wesley, 1999.

[7] JUSSI KASURINEN, O. T. A. K. S. *Software Test Automation in Practice:* Empirical Observations. Lappeenranta, Finland: [s.n.], 2009.

[8] DUSTIN, E.; RASHKA, J.; PAUL, J. *Automated Software Testing:* Introduction, Management, and Performance. 13ª. ed. Boston: Addison-Wesley, 2008.

[9] ROYER, J. C.; ARBOLEDA. *Model Driven and Software Product Line Engineering*. 1. ed. [S.l.]: Wiley-IEEE Press, 2012.

[10] SELIC, B. *Model Driven Development> Its essence and opportunities*. [S.l.]: IEEE Computer Society, 2006.

[11] OMG. *MDA Guide*. OMG. Boston. 2014.

[12] LUCRÉDIO, D. *Uma Abordagem Orientada a Modelos para Reutilização de Software*. São Paulo: Universidade de São Paulo, 2009.

[13] STHAL, T.; VOLTER, M. *Model-Driven Software Development - Technology,Engineering*. West Sussex: John Wiley & Sons, 2006.

[14] OMG. *MDA Guide*. [S.l.]: [s.n.], 2003.

[15] RÄTZMANN, M.; YOUNG, C. D. *Software Testing and Internationalization*. Salt Lake City: Lemoine, 2003.

[16] HEATH, S. *Embedded Systems Design*. 2ª. ed. Burlington: Newnes, 2003.

[17] MOREIRA, T. G. *Geração Automática de Código VHDL a partir de Modelos UML para*

Sistemas Embarcados em Tempo-Real. Porto Alegre: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2012.

[18] BROY, M. et al. *Engineering Automotive Software*. Munchen: Software & Syst. Eng., 2007.

[19] PINHEIRO, B. C. *Sistema de controle tempo real embarcado para automação de manobra de estacionamento*. Florianópolis: Universidade Federal de Santa Catarina, 2009.

[20] CARRO, L.; WAGNER, F. R. *Sistemas Computacionais Embarcados*. Porto Alegre: Universidade Federal do Rio Grande do Sul, 2003.

[21] WOLF, W. *Computers as Components:* Principles of Embedded Computing System Design. 2ª. ed. Burlington: Elsevier, 2008.

[22] PRESSMAN, R. S. *Engenharia de Software*. 6ª. ed. Rio de Janeiro: McGraw-Hill, 2006.

[23] SOMMERVILE, I. *Engenharia de Software*. 8ª. ed. São Paulo: Addison Wesley, 2007.

[24] MYERS, G. J. *The Art of Software Testing*. 2ª. ed. Hoboken: John Wiley & Sons, 2004.

[25] IEEE. Standard for software test documentation, 2008.

[26] RAMLER, R.; WOLFMAIER, K. *Economic Perspectives in Test Automation:* Balancing Automated and Manual Testing with Opportunity Cost. Shangai: Proceedings of the 2006 International Workshop on Automation of Software Test, 2006.

[27] THUMMALAPENTA, S. et al. *Automating Test Automation*. Zurich: 34th International Conference on Software Engineering, 2012.

[28] BRINGMANN, E.; KRÄMER, A. *Model-based Testing of Automotive Systems*. Lillehammer: International Conference on Software Testing, Verification, and Validation, 2008.

[29] XIAOCHUN, Z. et al. *A Test Automation Solution on GUI Functional Test*. Daejeon: International Conference on Industrial Informatics, 2008.

[30] LEITNER, A.; CIUPA, I.; MEYER, B. *Reconciling Manual and Automated Testing:* the AutoTest Experience. Waikoloa: Proceedings of the 40th Hawaii International Conference on System Sciences, 2007.

[31] CAETANO, C. *Melhores Práticas na Automação de Testes*. Rio de Janeiro: Revista Engenharia de Software Magazine, 2008.

[32] HOFFMAN, D. *Test Automation Architectures*. San Jose: International Quality Week

1999, 1999.

[33] SILVA, M. B. R. *Uma avaliação da abordagem MDA através de um estudo de caso*. Niterói: Universidade Federal Fluminense, 2005.

[34] KLEPPE, A. G.; WARMER, J. B.; BAST, W. *MDA Explained:* The Model Driven Architecture : Practice and Promise. Boston: Addison Wesley, 2003.

[35] LOPES, D. et al. *Mapping Specification in MDA:* From Theory to Practice. Geneva: Interoperability of Enterprise Software and Applications, 2006.

[36] CHAVES, R. A. *Aspectos e MDA:* Criando modelos executáveis baseados em aspectos. Florianópolis: Universidade Federal de Santa Catarina, 2004.

[37] OMG. *http:* //www.omg.org/mof/. [S.l.]: [s.n.], 2015.

[38] OMG. *Meta Object Facility Specification version 1.4*. [S.l.]: [s.n.], 2002.

[39] W, S. R. *Conceitos de Linguagens de Programação*. 5. ed. Porto Alegre: Bookman, 2003.

[40] GUEDES, G. T. A. *UML2 uma abordagem prática*. 2ª. ed. São Paulo: Novatec, 2011.

[41] OMG. *Unified Modeling Language Superstructure version 2.4.1*. [S.l.]: [s.n.], 2010.

[42] LARMAN, C. *Utilizando UML e Padrões*. Porto Alegre: Bookman, 2007.

[43] ANDRADE, V. C. *Transformação de modelos de diagrama de sequencia UML contemplando restrições de tempo e energia para rede de petri temporal*. curitba: Universidade Federal do Paraná, 2013.

[44] ZANCHETT, C. A. B. *Metamodelagem MOF e sua aplicação para modelagem de sistemas imunológicos artificias*. Florianópolis: Universidade Federal de Santa Catarina, 2005.

[45] TEPPOLA, S.; PARVIAINEN, P.; TAKALO, J. *Challenges in the Deployment of Model Driven Development*. Oulu: Fourth International Conference on Software Engineering Advances, 2009.

[46] CALIARI, G. L. P. *Transformações e mapeamentos da MDA e sua implementação em três ferramentas*. São Paulo: Universidade de São Paulo, 2007.

[47] CALIC, T.; DASCALU, S.; EGBERT, D. *Tools for MDA Software Development:* Evaluation Criteria and Set of Desirable Features. Las Vegas: Conference on Information Technology: New Generations, 2008.

[48] HUANG, Y. et al. *Design Validation Testing of Vehicle Instrument Cluster Using*

Machine Vision and Hardware-in-the-loop. Columbus: Conference on Vehicular Electronics and Safety, 2008.

[49] YONGFENG, Y.; BIN, L.; BENTAO, Z. *On Test Script Technique Oriented Automation of Embedded Software*. Los Angeles: World Congress on Computer Science and Information Engineering, 2009.

[50] OBELE, B. O.; KIM, D. *On an Embedded Software Design Architecture for Improving the Testability of In-Vehicle Multimidia Software*. Cleveland: IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2014.

[51] MOON, H. et al. *Automation Test Method for Automotive Embedded Software Based on AUTOSAR*. Porto: IEEE Fourth International Conference on Software Engineering Advances, 2009.

[52] PETERSEN, K. et al. Systematic mapping studies in software engineering. *Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering*. Bari: [s.n.]. 2008.

[53] FERREIRA, J. C. A. MDAI*:* Model based Design in Automobile Industry. *7th IEEE InternationalConference on Industrial Informatics*. Cardif: [s.n.]. 2009.

[54] GAILLIARD, G. et al. Transaction Level Modelling of SCA Compliant Software Defined Radio Waveforms and Platforms PIM/PSM. *Design, Automation & Test in Europe Conference & Exhibition*. Nice: [s.n.]. 2007.

[55] SINDICO, A.; NATALE, M. D.; SANGIOVANNI-VINCENTELLI, A. An Industrial System Engineering Process Integrating Model Driven Architecture and Model Based Design. *15th International Conference Proceedings*. Innsbruck: [s.n.]. 2012.

[56] GUTTMAN, M.; PARODI, J. *Real Life MDA:* Solving businessproblems with Model Driven Architecture. San Francisco: Elsevier, 2007.

[57] JAVED, A. Z.; STROOPER, P. A.; WATSON, G. N. Automated Generation of Test Cases Using Model-Driven Architecture. *Second International Workshop on Automation of Software Test*. Minneapolis: [s.n.]. 2007.

[58] CARROZZA, G. et al. IntegratingMDTinanIndustrialProcessintheAirTrafficControlDomain. *23rd International Symposium on Software Reliability Engineering Workshops*. Naples: [s.n.]. 2012.

[59] ALVES, E. L. G.; MACHADO, P. D. L.; RAMALHO, F. Automatic generation of built-

in contract test drivers. *Software and Systems Modeling Journal.* , New York, v. 13, 2014.

[60] WANG, F.; WANG, S.; JI, Y. An Automatic Generation Method of Executable Test Case Using ModelDriven Architecture. *International Conference on Innovative Computing, Information and Control*. Kaohsiung: [s.n.]. 2009.

[61] EDWARDS, C.; GRUNER, S. A new tool for URDAD to Java EE EJB Transformations. *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. East London: [s.n.]. 2013.

[62] BOLLATI, V. A. et al. Applying MDE to the (semi-)automatic development of model transformations. *Information and Software Technology*, n. 55, 2012.

[63] MOHAGHEGHI, P. et al. Where does model-driven engineering help? Experiences from three industrial cases, 2011.

[64] VALDERAS, P.; PELECHANO, V. A Survey of Requirements Specification in Model-Driven Development of Web Applications. *ACM Transactions on the Web*, v. 2, 2011.

[65] LONIEWSKI, G.; INSFRAN, E.; ABRAHÃO, S. A Systematic Review of the Use of Requirements Engineering Techniques in Model-Driven Development. *Model Driven Engineering Languages and Systems*. Oslo: [s.n.]. 2010.

[66] HEBIG, R.; BRENDRAOU, R. On the Need to Study the Impact of Model Driven Engineering on Software Processes. *Proceedings of the 2014 International Conference on Software and System Process*. Nanjing: [s.n.]. 2014.

[67] ALMEIDA, C. C. D. J.; OLIVEIRA, A. D. Qualitas*:* A Proposal of Process Model Development Software Driven Models. *Euro American Association on Telematics and Information Systems*. Valparaíso: [s.n.]. 2014.

[68] GACÍA-DÍAZ, V. et al. TALISMAN MDE: Mixing MDE Principles. *The Journal of Systems and Software*, v. 83, 2010.

[69] KRIOUILE, A.; ADDAMSSIRI, N.; GADI, T. An MDA Method for Automatic Transformation of Models from CIM to PIM. *American Journal of Software Engineering and Applications* , 2015.

[70] OMG. *Unified Modeling Language*. [S.l.]: [s.n.], 2015.

[71] OBJECT MANAGEMENT GROUP - OMG. *XML Metadata Interchange (XMI) Specification*. [S.l.]: [s.n.], v. 2.5.1, 2015.

[72] DUFFIELD, N. G. et al. Inferring link loss using striped unicast probes. *Proc. IEEE INFOCOM*. [S.l.]: [s.n.]. 2001. p. 915-923.

[73] VEHEL, J. L.; LUTTON, E.; C., T. *Fractals in engineering:* from theory to industrial applications. New York: Springer-Verlag, 1997.

[74] COMER, D. E. *Redes de Computadores e Internet*. 2ª. ed. São Paulo: Bookman, 2001.

[75] COATES, M. et al. Internet tomography. *IEEE Signal Processing Mag*, 19, May 2002. 47–65.

Anexos

ANEXO I

FORMULÁRIO DE DADOS DE TESTE
Data:___________________________
Responsável:____________________
Caso de Teste:___________________

LISTAS DE COMPONENTES
CLASSE IN: Classificação dos componentes de entrada
COMPONENTS IN: Componentes responsáveis por fornecer estímulos ao SUT.
BEHAVIOR IN: Comportamentos apresentados ou realizados por componentes de entrada.
STATES IN: Estados que um determinado componente de entrada pode assumir durante seu funcionamento.
PARAMETERS: Possíveis informações ou mensagens que possam ser transmitidos ao SUT por um dispositivo de entrada.
CLASSES OUT: Classificação dos componentes de saída.
COMPONENTS OUT: Componentes responsáveis por capturar um comportamento apresentado pelo SUT.
BEHAVIOR OUT: Componentes responsáveis por realizar um comportamento em resposta a uma determinada entrada.
STATES OUT: Estados que um determinado componente de saída pode assumir.
CLASSES CAP: Classificação dos componentes de captura
COMPONENTS CAP: Componentes responsáveis por capturar um comportamento apresentado pelo SUT.
BEHAVIOR CAP: Comportamentos ou ações que podem ser realizadas por um componente de captura.

ANEXO II

For textual notations a variant of the Backus-Naur Form (BNF) is often used to specify the legal formats. The conventions of this BNF are:

• All non-terminals are in italics and enclosed between angle brackets (e.g., <non-terminal>).

• All terminals (keywords, strings, etc.), are enclosed between single quotes (e.g., ‘or’).

• Non-terminal production rule definitions are signified with the ‘::=’ operator.

• Repetition of an item is signified by an asterisk placed after that item: ‘*’.

• Alternative choices in a production are separated by the ‘|’ symbol (e.g., <alternative-A> | <alternative-B>).

• Items that are optional are enclosed in square brackets (e.g., [<item-x>]).

• Where items need to be grouped they are enclosed in simple parenthesis; for example:

(<item-1> | <item-2>) *

signifies a sequence of one or more items, each of which is <item-1> or <item-2>. [70]

TRABALHOS PUBLICADOS PELO AUTOR

Trabalhos publicados ou aprovados para publicação pelo autor durante o programa de mestrado em Ciência da Computação.

1. Wilson H. Shirado, Márcio A. Moreira, Vanessa M. Leite, orientadora Jandira G. Palma, **Implementation Of Moprosoft In A Small Company With Mps.Br-F**, 8th IADIS INTERNATIONAL CONFERENCE (Qualis B4).
2. Wilson H. Shirado, Márcio A. Moreira, Cristina Yassue Morimoto, orientadora Jandira Guenka Palma, **Modelagem de Negócio como apoio ao Desenvolvimento Ágil de Software**, 12º CONTECSI (Qualis B4).
3. Wilson H. Shirado, Márcio A. Moreira, orientadores Jandira Guenka Palma e Sylvio Barbon Jr, **Estudo Comparativo entre Algoritmos das Transformadas Discretas de Fourier e Wavelet**, Revista Brasileira de Computação Aplicada, Outubro/2015, ISSN 2176-6649 (Qualis B5)
4. Wilson H. Shirado, Márcio A. Moreira, orientadores Jandira Guenka Palma, **Model Driven Architecture: Entendendo e aplicando os Conceitos**, Revista Engenharia de software Magazine, Outubro/2015, ISSN 1983127-7, (Qualis B5)
5. Wilson H. Shirado, Márcio A. Moreira, orientadores Jandira Guenka Palma, **Automação de Testes Funcionais Embarcados via Processamento Digital de Imagens**, Revista Engenharia de software Magazine, Agosto/2015, ISSN 1983127-7, (Qualis B5)
6. Wilson H. Shirado, Luiz Villela, Jandira G. Palma, **Levantamento de requisitos de software com Canvas,** Revista Engenharia de Software Magazine, Dezembro/2016, ISSN 1983127-7, (Qualis B5)