UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALEXANDRE AUGUSTO GIRON

Validação de transformações de modelos complexas

Maringá
2015

ALEXANDRE AUGUSTO GIRON

Validação de transformações de modelos complexas

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientadora: Prof[a]. Dr[a]. Itana Maria de Souza Gimenes

Maringá
2015

Dados Internacionais de Catalogação na Publicação (CIP)
(Biblioteca Central - UEM, Maringá, PR, Brasil)

Giron, Alexandre Augusto
G527v Validação de transformações de modelos complexas / Alexandre Augusto Giron. -- Maringá, 2015.
87 f. : il. color., figs., tabs.

Orientadora: Prof.ª Dr.ª Itana Maria de Souza Gimenes.
Dissertação (mestrado) - Universidade Estadual de Maringá, Centro de Tecnologia, Departamento de Informática, Programa de Pós-Graduação em Ciência da Computação, 2015.

1. Sistemas embarcados. 2. Transformações de modelos. 3. Engenharia de software - Testes. 4. Geração de casos de teste. I. Gimenes, Itana Maria de Souza, orient. II. Universidade Estadual de Maringá. Centro de Tecnologia. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 23.ed. 005.14
AMMA-002984

FOLHA DE APROVAÇÃO

ALEXANDRE AUGUSTO GIRON

Validação de transformações de modelos complexas

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação pela Banca Examinadora composta pelos membros:

BANCA EXAMINADORA

Profa. Dra. Itana Maria de Souza Gimenes
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Edson Alves de Oliveira Junior
Universidade Estadual de Maringá – DIN/UEM

Profa. Dra. Claudia Maria Lima Werner
Universidade Federal do Rio de Janeiro – COPPE/UFRJ

Aprovada em: 03 de junho de 2015.
Local da defesa: Sala 101, Bloco C56, *campus* da Universidade Estadual de Maringá.

DEDICATÓRIA

Aos meus pais, ao meu irmão, e à minha querida Fran.

AGRADECIMENTOS

Deixo aqui meus agradecimentos primeiramente a Deus, por ter me abençoado nesta caminhada. Agradeço de forma especial à minha família, por sempre me incentivar a conquistar o título de mestre. Agradeço também ao amor, carinho e apoio dado pela minha namorada Francielle.

Agradeço também à minha orientadora Dra. Itana Maria de Souza Gimenes, pelo apoio, comentários e sugestões que foram fundamentais no desenvolvimento deste trabalho.

Agradeço também de forma especial aos meus colegas de mestrado: Paulo, Mauricio, Romulo, Ariel, Ricardo e Marcinho, pelo apoio nos trabalhos das disciplinas, pela amizade e companheirismo que auxiliaram muito nessa conquista.

Os agradecimentos também devem ser dados aos demais colegas e professores das disciplinas que cursei, e também à secretária Maria Inês Davanço, pela paciência e auxílio durante o curso de mestrado.

Agradecimentos também à UEM de forma geral e à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro concedido durante quase todo o período de realização deste trabalho.

Validação de transformações de modelos complexas

RESUMO

A Engenharia Dirigida a Modelos (*Model-Driven Engineering* - MDE) apoia a evolução e o refinamento de modelos por meio de transformações em vários níveis de abstração. SyMPLES é uma abordagem de desenvolvimento para sistemas embarcados, que combina conceitos de MDE com Linha de Produto de *Software* (LPS). Essa abordagem possui um processo de transformação automatizado de modelos SysML para Simulink para guiar o desenvolvimento a partir de modelos rumo à implementação. A validação dessas transformações é importante para assegurar a qualidade dos modelos transformados. Neste trabalho é apresentada uma abordagem de validação baseada em teste funcional para transformações complexas, as quais são compostas por várias etapas distintas. A transformação da SyMPLES possui duas etapas, uma escrita em linguagem ATL e outra em linguagem Java. Essa transformação de modelos foi avaliada por meio da abordagem de validação utilizando duas técnicas de geração de casos de teste. A primeira utilizou uma LPS e a segunda o metamodelo SysML. A aplicação da abordagem identificou determinados tipos de erro. Utilizando políticas de geração e critérios de cobertura foi possível diminuir a quantidade de casos de teste gerados, o que possibilitou minimizar o esforço e o tempo do teste da transformação.

Palavras-chave: Sistemas embarcados, Validação de Transformação de modelos, Teste funcional, Geração de casos de teste.

Validation of complex model transformations

ABSTRACT

Model-Driven Engineering (MDE) supports model evolution and refining by means of transformations. SyMPLES is a development approach for embedded systems which is based on concepts of both Software Product Line (SPL) and MDE. This approach has a model transformation process from SysML to Simulink which guides the development from the models towards implementation. The validation of these transformations are important, to ensure the quality of the transformed models. Therefore, this work proposes a validation approach based on functional test for complex model transformations composed of several distinct steps. The SyMPLES transformation has two transformation steps, one written in ATL language and the other in Java languagem. This transformation has been evaluated by means of the validation approach using two test case generation techniques. The first used a SPL and the second used the SysML metamodel. The application of the approach identified certain kinds of error. By using generation policies and coverage criteria it was possible to reduce the amount of test cases generated, allowing to minimize the transformation test effort and time.

Keywords: Embedded Systems, Model Transformation Validation, Test Case Generation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

ATL: *ATLAS Transformation Language*
LPS: Linha de Produto de Software
MDE: *Model-Driven Engineering*
SyMPLES: *SysML-based Product Line Approach for Embedded Systems*
MDA: *Model-Driven Architecture*
MDD: *Model-Driven Development*
DSML: *Domain Specific Modeling Language*
PIM: *Platform-Independent Model*
PSM: *Platform-Specific Model*
CIM: *Computer-Independent Model*
UML: *Unified Modeling Language*
MBT: *Model-Based Testing*
QVT: *Query/View/Transformation*
VANT: Veículo Aéreo Não-Tripulado
XMI: *XML Metadata Interchange*
FM: *Feature Model*
BDD: *Binary Decision Diagram*
VAMT: *Validation Approach for Model Transformations*

SUMÁRIO

1

Introdução

1.1 Contextualização

Os Sistemas embarcados são sistemas computacionais que processam informações, normalmente não são diretamente visíveis ao usuário e são incorporados em um produto maior (Marwedel, 2003). Esses sistemas são populares atualmente e seus principais exemplos variam desde sistemas em pequenos dispositivos e celulares até em carros e eletrodomésticos. O processo de desenvolvimento de sistemas embarcados difere do desenvolvimento de sistemas comuns, pois eles possuem requisitos não-funcionais específicos, tais como consumo de potência reduzido, desempenho, integração otimizada de *hardware* e *software*, tempo-real, custo, *design* e tempo de projeto.

As abordagens de Engenharia Guiada por Modelos (*Model Driven Engineering* - MDE) e de Linha de Produto de *Software* (LPS) são alternativas que podem ser usadas para auxiliar o desenvolvimento de sistemas embarcados. Enquanto a MDE apoia a geração de aplicações e modelos por meio de transformações em diferentes níveis de abstração (Mellor, 2004), a LPS apoia o reúso de artefatos comuns a partir de um mesmo domínio, para customização de aplicações específicas (Linden et al., 2007).

Um dos objetivos de MDE é facilitar a geração de código a partir de modelos de especificação. O código pode ser gerado de forma automática a partir de progressivas transformações de modelos (Schmidt, 2006).

Uma LPS possui um núcleo de artefatos comuns, e assim produtos específicos são construídos reusando esse núcleo e adaptando-o conforme necessidades específicas. Essas adaptações, ou as formas nas quais os membros de uma família de produtos se diferenciam entre si, são as chamadas variabilidades (Belategi et al., 2010). Assim, as variabilidades

representam a diversificação de produtos. Os princípios de LPS podem complementar a MDE no processo de construção de sistemas embarcados, por meio de reúso e do gerenciamento de variabilidades.

Para apoiar o desenvolvimento de sistemas embarcados, combinando conceitos de MDE com LPS, foi proposta a abordagem *SysML-based Product Line Approach for Embedded Systems* (SyMPLES) (Silva et al., 2013),(Fragal et al., 2013). Essa abordagem utiliza modelos de alto nível de abstração, a partir dos quais se dá a resolução de variabilidades para configuração de produtos específicos. Os modelos configurados passam por transformações desde os níveis mais altos de abstração até modelos concretos próximos da geração de código. SyMPLES utiliza a linguagem SysML (Friedenthal et al., 2011). Essa linguagem é uma extensão da UML com apoio à análise, especificação, projeto, verificação e validação de sistemas complexos e dinâmicos.

Com o objetivo de transformar os modelos em direção à implementação (código-fonte), a abordagem SyMPLES possui um processo de transformação de modelos SysML para Simulink (MathWorks, 2014). Essa transformação de modelos utiliza um modelo SysML gerado no contexto da abordagem SyMPLES, e o transforma em um modelo Simulink correspondente, o qual é manipulado pelo MATLAB/Simulink, que é um ambiente de modelagem e simulação de sistemas embarcados por meio de um conjunto de blocos funcionais, muito utilizado para desenvolvimento de sistemas embarcados (Fragal, 2013).

1.2 Motivação

As técnicas de validação das transformações de modelos podem ser usadas para assegurar a qualidade das aplicações geradas com transformações (Fleurey et al., 2004). Se os modelos são derivados de forma automática por meio de uma transformação, então a qualidade desses modelos vai depender da qualidade do processo de transformação (Küster e Abd-El-Razik, 2006).

Uma transformação de modelos pode ser implementada como um programa escrito em uma linguagem de programação, como por exemplo a linguagem JAVA, ou por meio de uma linguagem de transformação, como por exemplo a ATL (Jouault e Kurtev, 2006). Devido a este fato, tanto o teste funcional como o teste estrutural podem ser utilizados para o teste de transformações. A diferença básica entre os dois é que no teste funcional a especificação do programa é considerada, enquanto que no teste estrutural são considerados os aspectos internos do programa (Sen et al., 2009) (González e Cabot, 2012).

Na transformação de modelos da abordagem SyMPLES constatou-se a necessidade de oferecer um conjunto maior de recursos da linguagem SysML (Fragal, 2013). Além disso, a transformação não foi validada; ela foi avaliada somente em um exemplo de aplicação. Este exemplo foi desenvolvido para um subsistema da placa Yapa 2, responsável pelo controle de pilotagem automática, no contexto do Instituto Nacional de Ciência e Tecnologia para Sistemas Embarcados Críticos (INCT-SEC). Os resultados desta avaliação mostraram que o uso da transformação SysML para Simulink facilita a especificação do sistema em uma linguagem de mais alto nível e posterior simulação e geração de código da aplicação. Assim, para melhorar a abordagem SyMPLES, é necessário que seu processo de transformação seja validado para casos gerais para fornecer maior confiança e qualidade dos modelos gerados. A validação poderia ser realizada por meio de uma prova formal da transformação de modelos que requer técnicas de verificação formal. Outra alternativa, é a validação por meio de técnicas de teste de *software*, estrutural ou funcional, que é muito utilizada na indústria (Küster e Abd-El-Razik, 2006).

O teste de transformações de modelos é significativamente diferente do teste de *software* comum. Isso ocorre principalmente devido à estrutura complexa dos modelos de especificação, os quais podem possuir diferentes tipos de elementos e diferentes arranjos entre os mesmos (Tiso et al., 2012). Além disso, uma transformação de modelos pode ser implementada por meio de diferentes linguagens e em diferentes etapas de transformação, aumentando a sua complexidade. Algumas dessas linguagens são declarativas e baseadas em regras e isso dificulta a realização do teste estrutural.

A transformação de modelos SysML para Simulink é um exemplo de transformação em duas etapas (Fragal et al., 2013), cada uma construída com uma linguagem diferente. A primeira etapa é construída em linguagem ATL e a segunda em linguagem Java. ATL é uma linguagem essencialmente declarativa para desenvolver transformações de modelos. Dessa forma, o teste funcional é mais adequado para este caso.

Por outro lado, modelos frequentemente podem ter vários elementos distintos, definidos no metamodelo correspondente, o que dificulta o teste funcional da transformação, pois muitos casos de teste podem ser gerados. Isso ocorre pois a natureza complexa dos modelos dificulta a construção de casos de teste, que são os modelos de entrada para o teste (Tiso et al., 2012).

Dessa forma, uma solução é a construção manual ou automática de um conjunto reduzido de modelos e ao mesmo tempo seguindo um critério de cobertura mais abrangente possível (Tiso et al., 2012), para aplicação do teste.

1.3 Objetivos

Este trabalho de mestrado tem por objetivo geral propor uma abordagem de validação baseada em teste funcional de *software* para transformações de modelos complexas, e utilizá-la para validação da transformação SysML para Simulink, no contexto da abordagem SyMPLES. Os objetivos específicos são:

- Utilizar a técnica de geração de casos de teste a partir de metamodelos para validar a transformação, considerando o comportamento desta para casos genéricos.
- Aplicar a abordagem de validação proposta para a validação da transformação SysML para Simulink, avaliando-a e comparando-a às técnicas utilizadas para geração dos casos de teste e também comparando a proporção de erros encontrados na transformação por casos de teste gerados.

1.4 Organização do texto

O texto está organizado como segue: o Capítulo 2 apresenta uma revisão bibliográfica sobre MDE e LPS, abordagem SyMPLES, linguagem SysML, Simulink e ATL, e técnicas de validação específicas para transformações de modelos. O Capítulo 3 descreve a abordagem de validação proposta, o Capítulo 4 explica como a abordagem de validação foi aplicada para o teste da transformação SysML para Simulink, bem como os resultados obtidos, e, por fim, constam as conclusões, as referências e os apêndices.

2

Revisão Bibliográfica

2.1 Considerações Iniciais

Abordagens como MDE e LPS podem ser utilizadas no contexto do desenvolvimento de sistemas embarcados. Utilizando esses conceitos, foi proposta a abordagem SyMPLES para auxílio no desenvolvimento desses sistemas. Assim, este capítulo apresenta as características de LPS e MDE, bem como os conceitos básicos das linguagens SysML, ATL e Simulink. A SyMPLES e o estado da arte da validação de transformações de modelos são descritos, pois constituem os princípios fundamentais para esta dissertação de mestrado.

2.2 Linha de Produto de Software (LPS)

As LPS (Linden et al., 2007), (Belategi et al., 2010) são utilizadas devido a sua abordagem de desenvolvimento sistemática, que possibilita reduzir o tempo de lançamento do produto (*time-to-market*). Surgiram em meados dos anos 90 e têm como foco a construção de um conjunto de produtos similares de um domínio específico.

A abordagem de LPS é baseada em dois conceitos principais: customização em massa e plataformas comuns (Pohl et al., 2010). A customização em massa visa a produção em grande escala de produtos adaptáveis às necessidades específicas, enquanto o conceito de plataformas comuns apoia a construção de um núcleo comum de recursos, a ser utilizado como base para a construção de diversos produtos, tecnologias ou processos. A diferenciação entre produtos a partir de um núcleo comum é estabelecida por meio da identificação de características variáveis (variabilidades) de um produto, de acordo

com requisitos de domínios específicos. Esse processo é conhecido como gerência de variabilidades (Belategi et al., 2010). Atualmente existem técnicas e ferramentas para auxiliar esse processo, sendo algumas delas explicadas a seguir.

2.2.1 Análise Automatizada em Linha de Produto

Modelos de Características (MC) são modelos que auxiliam a gerência de variabilidades em LPS. Um MC representa os possíveis produtos de uma LPS. A análise automatizada de uma LPS pode utilizar um MC para obter algumas informações sobre a LPS sob análise, tais como: número de possíveis produtos, número de produtos que seguem uma restrição, custo mínimo de configuração, entre outras.

Um MC pode ser representado utilizando a ferramenta pure::variants (Beuche, 2012). Nessa ferramenta, ele pode ser definido como um *Variant Description Model* (VDM), no qual as características são listadas e podem ser selecionadas para que o produto seja instanciado. Um exemplo de VDM disponível na pure::variants pode ser visualizado na Figura 2.1. Nesse exemplo, apenas o *WindSensor* foi selecionado e isso significa que o produto será instanciado apenas com essa característica.

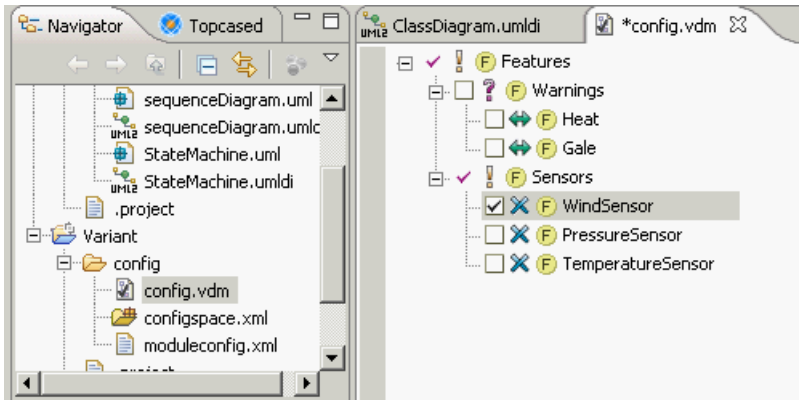

Figura 2.1: Exemplo de *Variant Description Model*

Para efetuar uma análise automatizada de uma LPS, geralmente são requeridos dois passos essenciais (Benavides et al., 2007):

- Transformar um MC em uma determinada representação lógica;
- Utilizar um algoritmo de decisão para extrair informações sobre a linha de produto. Alguns algoritmos são conhecidos como *Satisfiability Modulo Theories (SMT) Solvers* (Büttner et al., 2012).

Para a representação lógica de MC, várias técnicas foram propostas. As três mais utilizadas para análise em LPS são: *Constraint Satisfaction Problem* (CSP), *Boolean Satisfiability Problem* (SAT) e *Binary Decision Diagram* (BDD). Após um MC ser transformado em uma representação lógica, um algoritmo pode obter as informações desejadas na análise da LPS.

Em Benavides et al. (2007) e Mendonca et al. (2009) foram propostas ferramentas para análise automatizada para LPS. Além disso, testes de desempenho com a utilização das técnicas citadas também foram realizados nesses trabalhos, mostrando que os BDDs são mais eficientes em termos de tempo de execução.

Um BDD é uma estrutura de dados acíclica para representação de uma função booleana, composta por nós de decisão e dois tipos de nós terminais (0 e 1) (Benavides et al., 2007). Cada nó representa uma variável booleana e possui dois nós-filho, os quais representam uma atribuição à variável booleana, entre 0 e 1. Assim, todos os caminhos que levam a um nó 1-terminal são as configurações das variáveis que tornam a função booleana verdadeira.

Um exemplo de BDD pode ser visualizado na Figura 2.2. Ele é uma representação de uma função booleana S, que recebe três variáveis (x1, x2 e x3), de acordo com a Tabela-Verdade na mesma figura.

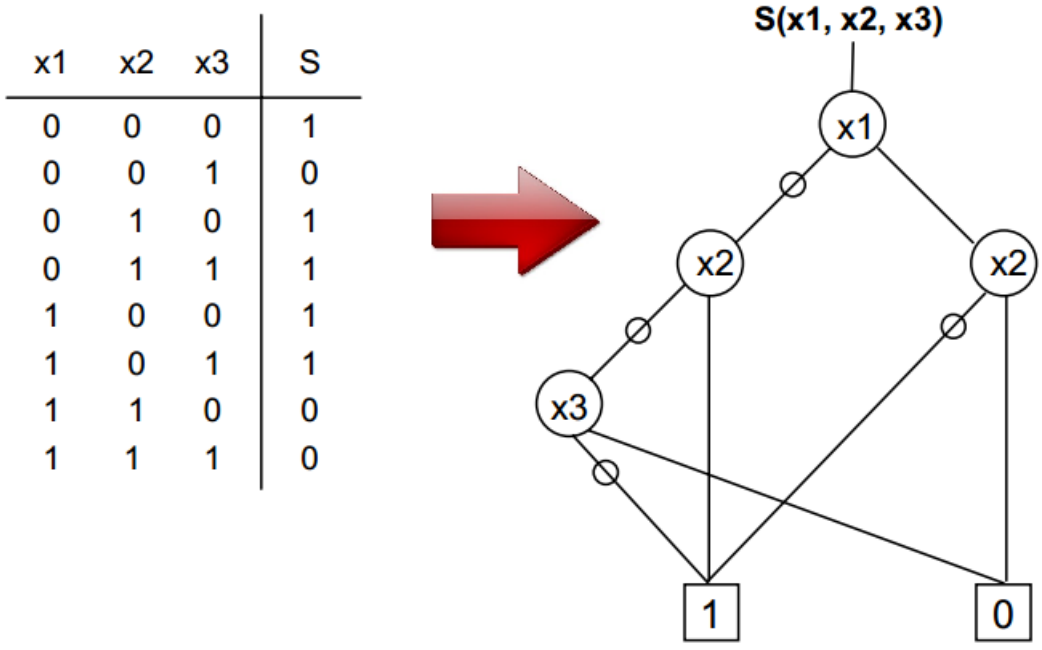

Figura 2.2: Exemplo de função booleana representada por um diagrama de decisão binária (BDD).

Na Figura 2.2, a função S pode ser resolvida percorrendo a árvore binária de acordo com o valor da variável. Se o valor for igual a 0, o caminho segue para a esquerda; se o valor for 1, o caminho segue para a direita. Por exemplo, se os valores para as variáveis forem todos iguais a 0, o retorno da função deve ser 1. No diagrama correspondente, percorrendo a árvore sempre à esquerda, obteremos o valor terminal igual a 1. Assim, todos os caminhos na árvore que levarem ao nó terminal 1 satisfazem a função booleana especificada na tabela.

Com a especificação de uma LPS por meio de um modelo de *features*, é possível utilizar um BDD para representar esse modelo e extrair então informações da LPS, como por exemplo, a quantidade de configurações válidas de produtos, respeitando as restrições que eventualmente aparecem em uma LPS. Exemplos de restrições, denotadas com estereótipos da abordagem SyMPLES, são: "mandatory", "alternative_XOR", entre outras. Na prática, as restrições na LPS são mapeadas para restrições no BDD. Um MC simplificado para exemplo com uma *core feature* composta por um grupo de três variantes, mostrada na Figura 2.3, foi criado por meio da ferramenta SPLOT (Mendonca et al., 2009).

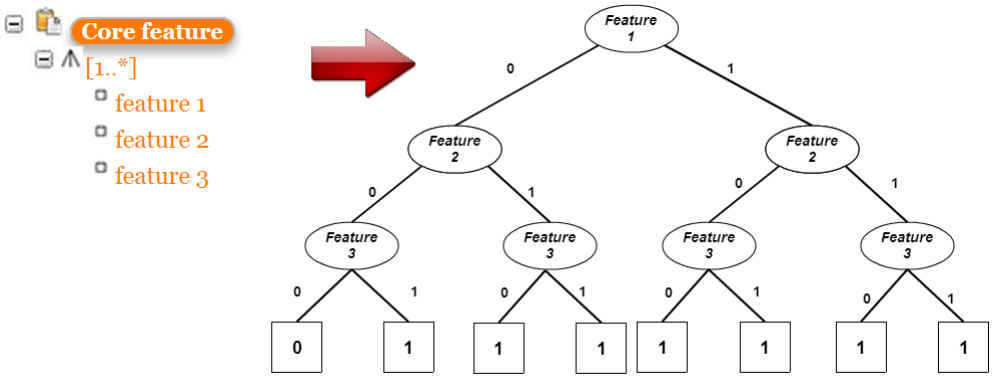

Figura 2.3: Exemplo de criação de um BDD com a análise de uma LPS.

A parte mais a direita da Figura 2.3 mostra o diagrama BDD correspondente ao MC apresentado, considerando cada *feature* como uma variável de tabela-verdade, e que a LPS é configurada pela operação lógica OU entre as variantes. Assim, um produto válido é uma configuração que satisfaça a condição lógica OU na tabela-verdade, ou um caminho que leve a um nodo final igual a 1 no BDD. Todas as soluções que satisfazem essa condição representam o total de configurações possíveis e válidas para a LPS, e podem ser determinadas por meio da contagem dos caminhos que levam aos nodos iguais a 1. Nesse exemplo, a quantidade total de configurações possíveis é igual a **7**, pois é a quantidade de caminhos no BDD que levam a um nodo final igual a 1.

2.3 Model-Driven Engineering (MDE)

A MDE pode ser descrita como uma abordagem que guia o desenvolvimento apoiado por técnicas de transformação sobre os modelos de especificação do sistema. As técnicas de MDE fornecem uma solução para problemas como: complexidade de implementação em plataformas específicas e representação de conceitos do domínio de aplicação que as linguagens de implementação possuem (Schmidt, 2006). Essa solução inclui modelos e esquemas do sistema com alto nível de abstração.

A MDE fez surgir outras abordagens guiadas por modelos, com nomenclatura semelhante, que são a *Model-Driven Architecture* (MDA) e a *Model-Driven Development* (MDD). A primeira é relacionada com a definição de padrões de interoperabilidade entre as tecnologias de desenvolvimento e as transformações, e a segunda pode ser descrita como um processo de desenvolvimento. Esse processo é relacionado com a construção de sistemas de forma flexível e com simplificação das notações dos modelos (Kent, 2002).

Além disso, a MDE combina dois conceitos básicos, que são as DSMLs (*Domain Specific Modeling Languages*) e os motores e geradores de transformação (Del Fabro e Valduriez, 2007; Kent, 2002). Uma DSML é descrita por meio de um metamodelo e é usada para formalizar a estrutura da aplicação, bem como auxiliar desenvolvedores na definição de relacionamentos conceituais no domínio de uma aplicação. Um motor de transformação analisa aspectos dos modelos e, a partir disso, gera uma gama de artefatos, como por exemplo um conjunto de modelos alternativos ou código de implementação.

De acordo com o nível de abstração, os modelos podem ser classificados como CIM (*Computer-Independent Model*), PIM (*Platform-Independent Model*), e PSM (*Platform Specific Model*). Os modelos CIM são modelos de alto nível de abstração, e os modelos PIM contêm especificação com caráter independente de plataforma. Os modelos PSM dependem de uma plataforma específica e podem ser transformados para código de aplicação (Almeida, 2008).

A transformação de modelos da MDE pode ser de três tipos principais, de acordo com o modelo de entrada e com o modelo de saída. Assim, as transformações de modelos podem ser em níveis de abstração, de refinamento ou baseadas em engenharia reversa (Almeida, 2008). Essa classificação pode ser visualizada na Figura 2.4. A transformação em níveis de abstração modifica o tipo do modelo (CIM para PIM por exemplo), enquanto que a transformação de refinamento modifica o modelo, mas mantém o tipo do modelo de saída igual (PIM para PIM, por exemplo). O terceiro tipo, transformação baseada em engenharia reversa, tem o objetivo de recuperar informações dos modelos de níveis

de abstração mais baixos, e tem maior utilidade no contexto da manutenção de *software* evolutivo (Pressman, 2010).

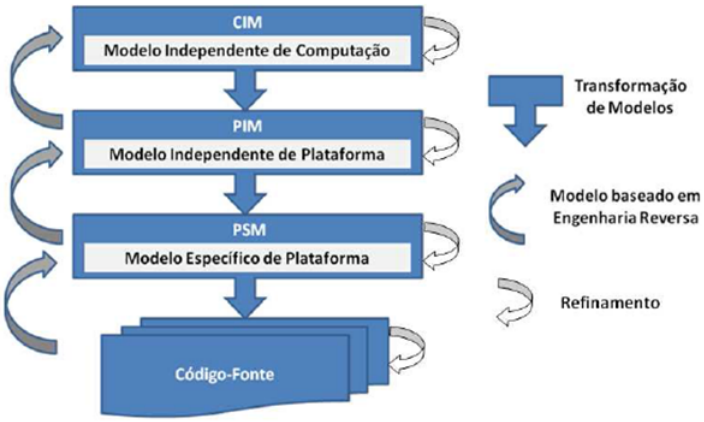

Figura 2.4: Tipos de transformações de modelos (extraído de Almeida (2008))

Czarnecki e Helsen (2003) fornecem outra classificação para as transformações de modelos de acordo com o tipo dos modelos de entrada e de saída. Em transformações de modelos *Model-to-Code* os modelos de entrada são modelos escritos em alguma linguagem de modelagem, e os modelos de saída são arquivos de código-fonte escritos por meio de alguma linguagem de programação. Nas transformações *Model-to-Model* tanto os modelos de entrada quanto os de saída são escritos por meio de alguma linguagem de modelagem, como por exemplo a linguagem SysML.

No contexto da MDE, algumas linguagens e ferramentas podem ser utilizadas para especificação e desenvolvimento de sistemas embarcados, entre elas a linguagem SysML e Simulink. Linguagens de transformação de modelos também são importantes para a MDE, tais como QVT e ATL (Jouault et al., 2006). A seguir, as linguagens SysML, Simulink e a ATL serão descritas pois são utilizadas nesta dissertação.

2.3.1 SysML

SysML é uma linguagem para modelagem voltada à construção e especificação de sistemas complexos e dinâmicos (Friedenthal et al., 2011). Ela é descrita como uma linguagem de modelagem visual, independente de metodologia e de ferramenta, que foi concebida como um perfil da linguagem UML (*Unified Modeling Language*) (OMG, 2011).

De acordo com Friedenthal et al. (2011), o fato da SysML ser voltada a sistemas complexos significa que ela deve representar vários elementos distintos, tais como *hardware*, *software*, dados, pessoal, procedimentos, instalações e sistemas naturais. Um conceito

importante na linguagem SysML é a **Modelagem com Blocos**, em que um bloco é uma unidade modular de estrutura que pode ser utilizado tanto para definir um sistema quanto para definir um componente do sistema, ou ainda, para definir um objeto ou um dado que flui por meio do sistema.

A linguagem SysML pode ser classificada como um perfil da UML, pois ela incorpora ao seu conjunto de diagramas grande parte dos diagramas da UML. Os diagramas SysML podem ser classificados da seguinte forma (Friedenthal et al., 2011):

- **Diagramas Estruturais**: pertencem à essa categoria o diagrama de pacotes que é igual ao da linguagem UML e os diagramas de Definição de blocos e Interno de bloco, que são modificados em relação à UML. Além disso a SysML fornece o diagrama Paramétrico para representar restrições sobre valores de determinadas propriedades do sistema, tais como desempenho e confiabilidade.

- **Diagramas Comportamentais**: nessa classificação constam os diagramas de sequência, estados e de casos de uso, e são os mesmos da UML. O diagrama de atividades também está nessa classificação, mas ele é modificado em relação à UML.

- **Diagramas Transversais**: o diagrama de Requisitos é o único diagrama dessa classificação. Esse diagrama é exclusivo da linguagem SysML e representa os requisitos do sistema de forma textual e seu relacionamento com outros requisitos. Um dos objetivos do uso desse diagrama é fornecer o conceito de rastreabilidade, auxiliando na gestão dos requisitos.

2.3.2 Simulink

O MATLAB/Simulink é um ambiente que permite a modelagem de um sistema por meio de blocos funcionais. O MATLAB/Simulink é uma ferramenta industrial muito utilizada no contexto dos sistemas dinâmicos e embarcados (Reicherdt e Glesner, 2012). O Simulink utiliza como base o ambiente MATLAB, e vários são os exemplos de sistemas que são desenvolvidos com o MATLAB/Simulink, como por exemplo robôs, carros e aviões. Apesar do Simulink ser um ambiente para modelagem de blocos funcionais, neste trabalho o ambiente Simulink é tratado como linguagem Simulink.

Na modelagem de blocos funcionais do Simulink, a comunicação é dada por um fluxo de dados entre as portas dos blocos, descrevendo o comportamento do sistema ou do subsistema. Para a modelagem, o Simulink fornece um conjunto de blocos pré-definidos, que são organizados em bibliotecas. Cada biblioteca é descrita a seguir:

- *Sources*: possui blocos geradores de sinais. Exemplos são os blocos *Signal Generator* e *Sine wave*.
- *Sinks*: possui blocos que tem função de exibir ou escrever sinais. Exemplos são os blocos *Scope* e *XY Graph*.
- *Discrete*: blocos que descrevem componentes discretos no tempo. Exemplos são os blocos *Discrete State-Space* e *Discrete Filter*.
- *Continuous*: conjunto de blocos que descrevem funções lineares. Exemplos: *Integrator* e *Derivative*.
- *Nonlinear*: possui blocos que descrevem funções não-lineares. Exemplos são os blocos *Relay* e *Quantizer*.
- *Math*: conjunto de blocos que representam funções matemáticas. Exemplos: *Sum* e *Dot Product*.
- *Functions & Tables*: constituída de blocos que descrevem funções gerais e operações de tabelas. Exemplos: *Look-up Table* e *Fcn*.
- *Signal & Systems*: conjunto de blocos que permitem criar subsistemas e módulos, adicionar multiplexadores e demultiplexadores, criar portas IN/OUT, armazenagem de dados e outras funções. Exemplos: *Subsystem* e *Data Store Memory*.

Apesar de serem bibliotecas específicas, elas são integradas com as bibliotecas do ambiente MATLAB e também é permitido criar novos blocos ou bibliotecas customizadas para serem utilizadas. Vale ressaltar que o Simulink classifica os blocos em dois tipos básicos: blocos virtuais e os não-virtuais. A diferença básica entre eles é o fato de que os blocos virtuais não tem papel ativo em uma simulação do sistema no ambiente MATLAB/Simulink, e então tem papel apenas de ajudar na organização gráfica do modelo. Por outro lado, os blocos não-virtuais interferem no processo de simulação do sistema.

2.3.3 ATL

A ATL é uma linguagem para desenvolvimento de transformação de modelos por meio da utilização de regras de transformação. É uma linguagem popular por ter uma sintaxe simples e por trabalhar com metamodelos no padrão Ecore, que é baseado no

Meta-Object Facility (MOF) (Jouault e Kurtev, 2006). A linguagem ATL foi utilizada para a transformação de modelos da abordagem SyMPLES.

Por meio de um motor de transformação, uma transformação é compilada e executada. A execução das transformações com a linguagem ATL ocorre por meio de uma máquina virtual chamada de *ATL Virtual Machine* (VM) (Jouault et al., 2006).

A ATL é caracterizada como uma linguagem híbrida, pois é composta por regras declarativas e também por construtores imperativos, embora os elementos declarativos para as transformações sejam mais utilizados. Os principais tipos de regras ATL são:

- *Matched*: as regras desse tipo são declarativas, e são aplicadas nos elementos dos metamodelos de entrada. São executadas sequencialmente e apenas uma vez.
- *Lazy*: essas regras são similares às anteriores, mas devem ser referenciadas por outras regras para executarem. Como essa regra pode ser aplicada várias vezes para cada correspondência entre elementos dos metamodelos, em geral sua utilização se torna mais conveniente.

Um exemplo de transformação simples utilizando ATL é apresentado a seguir. Esse exemplo foi extraído da documentação da linguagem ATL. A transformação recebe modelos de entrada de acordo com o metamodelo *Families*, e os transforma em modelos de saída de acordo com o metamodelo de saída *Persons*. Os metamodelos podem ser visualizados na Figura 2.5.

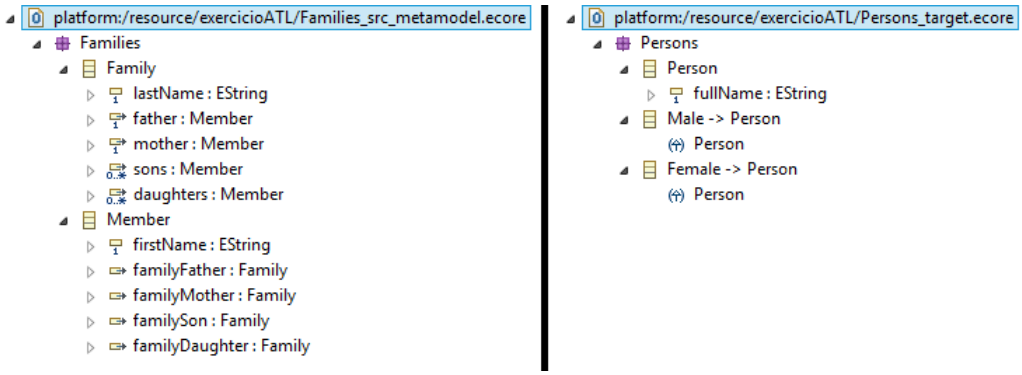

Figura 2.5: Metamodelo *Families* à esquerda e metamodelo *Persons* à direita

Para realizar a transformação, foram utilizadas regras ATL e *helpers*, que são rotinas ou métodos de processamento sobre tipos. As regras usadas podem ser visualizadas na Figura 2.6, responsáveis por realizar as ações correspondentes, a cada ocorrência de um elemento no modelo. Se aparecer um elemento do tipo *Member*, de acordo com sua característica,

```
45 -- RULES
46 --
47 rule Member2Male {
48     from
49         s: Families!Member (
50             not s.isFemale()
51         )
52     to
53         t: Persons!Male (
54             fullName <- s.firstName + ' ' + s.familyName
55         )
56 }
57
58 rule Member2Female {
59     from
60         s: Families!Member (
61             s.isFemale()
62         )
63     to
64         t: Persons!Female (
65             fullName <- s.firstName + ' ' + s.familyName
66         )
67 }
```

Figura 2.6: Regras ATL de transformação

uma regra diferente é aplicada. Assim, haverá a separação entre os membros de uma família, como resultado da transformação.

Na parte superior da Figura 2.7 consta o modelo de entrada para ser transformado. Cada elemento desse modelo representa um membro de uma determinada família. Executando a transformação é produzido um modelo de saída, contendo os elementos transformados de acordo com o metamodelo de saída. O resultado da transformação, que classifica cada membro de cada família, pode ser visualizado na parte inferior da Figura 2.7.

2.4 SyMPLES

SyMPLES é uma abordagem que combina conceitos de MDE com LPS para o desenvolvimento de sistemas embarcados. Nela os modelos são especificados em SysML com extensões para representação de variabilidades.

A SyMPLES pode ser dividida em dois contextos: engenharia de domínio, na qual estão as atividades relacionadas com a LPS, como por exemplo o gerenciamento de variabilidades; e engenharia de aplicação, na qual está a transformação de modelos. Essa representação genérica pode ser visualizada na Figura 2.8, adaptada de Fragal et al. (2013).

A abordagem utilizou o mecanismo de *profiling* para criação de dois perfis de extensão da linguagem SysML, os quais são descritos a seguir (Silva, 2012):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Families">
    <Family lastName="Young">
        <father firstName="Jim"/>
        <mother firstName="Cindy"/>
        <sons firstName="Angus"/>
        <sons firstName="Malcolm"/>
    </Family>
    <Family lastName="Sailor">
        <father firstName="Peter"/>
        <mother firstName="Jackie"/>
        <sons firstName="David"/>
        <daughters firstName="Kelly"/>
    </Family>
</xmi:XMI>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Persons">
  <Male fullName="Jim Young"/>
  <Male fullName="Angus Young"/>
  <Male fullName="Malcolm Young"/>
  <Male fullName="Peter Sailor"/>
  <Male fullName="David Sailor"/>
  <Female fullName="Cindy Young"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Kelly Sailor"/>
</xmi:XMI>
```

Figura 2.7: Modelo de entrada (parte superior) para a transformação e o modelo de saída correspondente (parte inferior).

- SyMPLES *Profile for Functional Blocks* (SyMPLES-ProfileFB): é um perfil para blocos funcionais que utiliza um grupo de estereótipos para fornecer uma semântica adicional aos blocos SysML.

- SyMPLES *Profile for Representation of Variability* (SyMPLES-ProfileVar): baseado na abordagem *SMarty* (Oliveira Junior et al., 2010), esse perfil representa as variabilidades de uma LPS a partir de um conjunto de estereótipos e valores agregados aos elementos dos diagramas SysML.

A SyMPLES também define dois processos de apoio ao desenvolvimento de LPS, os quais definem um grupo de atividades e diretrizes para sua execução. Esses processos são descritos a seguir:

- SyMPLES *Process for Product Lines* (SyMPLES-ProcessPL): processo de desenvolvimento de LPS com a função de auxiliar a criação de seus artefatos no domínio dos sistemas embarcados. Esse processo é composto de atividades independentes de ferramenta que auxiliam na criação dos artefatos de uma LPS.

- SyMPLES *Process for Identification of Variabilities* (SyMPLES-ProcessVar): também baseado no processo da abordagem *SMarty*, este processo tem o objetivo de apoiar as atividades de identificação e representação de variabilidades de uma LPS, bem como auxiliar na configuração dos produtos.

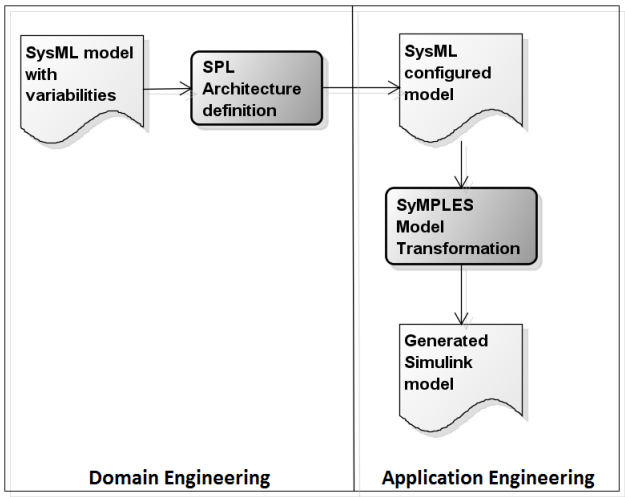

Figura 2.8: Representação da abordagem SyMPLES

No perfil de blocos funcionais da abordagem SyMPLES foram criados estereótipos sobre os elementos da linguagem SysML para mapeá-los aos principais tipos de blocos funcionais. Seguindo-se ao trabalho de Silva (2012) que concentrou-se na engenharia de domínio de SyMPLES, Fragal (2013) desenvolveu uma técnica para transformação de modelos SysML para modelos Simulink correspondentes.

2.4.1 Transformação de Modelos SysML para Simulink - versão inicial

O objetivo da transformação é fornecer um refinamento dos modelos SysML configurados com os estereótipos SyMPLES, de forma automatizada. Assim, os modelos gerados podem ter maior proximidade com relação a implementação do sistema especificado.

A transformação de modelos SysML para Simulink é dividida em três etapas: configurar um modelo SysML, executar a transformação escrita em linguagem ATL e gerar blocos funcionais, conforme mostra a Figura 2.9, adaptada de (Fragal et al., 2013).

Para transformar modelos SysML configurados em modelos Simulink correspondentes, é necessário realizar as três etapas, que são detalhadas a seguir:

- **Configurar um modelo SysML**: essa etapa gera o modelo SysML configurado que será transformado. O modelo pode ser composto de três diagramas: Definição de Blocos, Interno de Bloco, Máquina de Estados.

- **Executar a transformação ATL**: essa etapa utiliza regras ATL para selecionar informações relevantes do modelo SysML, como atributos de elementos, estereótipos e dados gráficos (posições dos elementos e tamanho). Cada elemento com informações

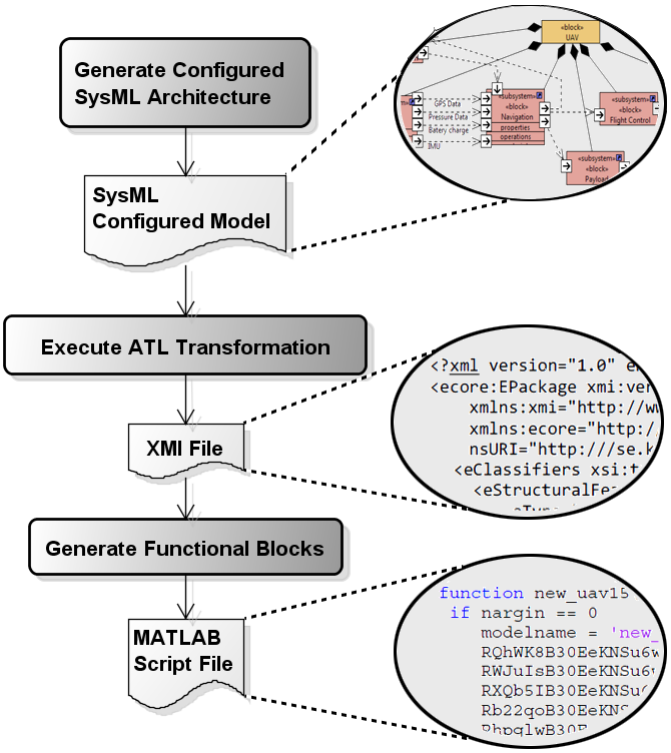

Figura 2.9: Transformação de modelos da SyMPLES (adaptada de (Fragal et al., 2013))

relevantes será mapeado para um bloco funcional, e então um modelo intermediário XMI é gerado.

- **Gerar blocos funcionais**: essa etapa gera um *script* executável para a plataforma MATLAB. As entradas dessa etapa são: o modelo intermediário XMI produzido pela etapa anterior e o arquivo principal do modelo SysML, nomeado como arquivo UML. O arquivo UML deve ser utilizado, pois ele contém valores referenciados pelo modelo intermediário, e principalmente para obter os estereótipos do perfil de blocos funcionais SyMPLES, os quais são usados para o mapeamento para gerar os blocos funcionais. Após a execução desse *script* o modelo Simulink é gerado. Um exemplo desse mapeamento é quando um bloco SysML com o estereótipo ««demux»»for processado, um bloco Simulink do tipo "demux" será especificado no *script*. Após a execução desse *script* o modelo Simulink é gerado.

2.4.2 Transformação de Modelos SysML para Simulink - versão estendida

Inicialmente, a transformação de modelos da abordagem SyMPLES não possuía suporte para o diagrama Paramétrico. Neste trabalho, ela foi estendida para incluir esse diagrama

pois ele é importante por definir restrições sobre valores de propriedades do sistema e inclusive requisitos não-funcionais. Essas restrições são frequentemente utilizadas no desenvolvimento de sistemas embarcados, sendo um exemplo típico a restrição de tempo-real (Marwedel, 2003). A seguir são apresentados os detalhes sobre a incorporação deste diagrama na transformação, bem como a estrutura e um exemplo de aplicação de transformação.

Características do Diagrama Paramétrico

A utilização do diagrama paramétrico permite a captura de parâmetros de análise, tais como desempenho, medidas de efetividade (*Measure of Effectiveness* - MOE's), confiabilidade, custo, entre outros. Além disso, alguns tipos de análise podem ser realizados com o uso de diagramas paramétricos, tais como: análise sensitiva, para determinar quais valores de propriedades têm impacto nos requisitos do sistema; estudos comparativos, com soluções alternativas para o sistema; e análise para otimização, com o uso de diagramas paramétricos de alto nível que possuem um conjunto de restrições no sistema, e com diagramas paramétricos de baixo nível especificando cada restrição de forma específica (Friedenthal et al., 2011).

Normalmente esses diagramas não existem por si sós, pois podem ser criados a partir de um diagrama de definição de blocos. Assim, o diagrama paramétrico serve como um meio para integrar os modelos das fases de especificação e projeto com modelos na fase de análise (Linhares et al., 2006).

Um exemplo simplificado de um diagrama paramétrico, extraído do manual de referência da linguagem SysML (OMG, 2012), é mostrado na Figura 2.10. Esse diagrama ilustra as relações entre as propriedades de um sistema de controle de combustível, entre elas a *Fuel Demand* (demanda da bomba injetora de combustível) e *Fuel Pressure* (pressão), que são usadas para o cálculo da taxa do fluxo de combustível, representado pelo bloco *Fuel Flow Rate*. Além de mostrar essas relações, o diagrama paramétrico usualmente contém blocos do tipo restrições de propriedade (*constraint property*), que representam as restrições sobre as propriedades por meio de funções matemáticas. Assim, pode-se perceber que o diagrama paramétrico auxilia na identificação e representação das restrições que podem acabar impactando nos requisitos do sistema.

Diagramas paramétricos podem, inclusive, mostrar requisitos não-funcionais do sistema (Halim et al., 2012). Apesar disso, esses diagramas requerem um esforço significativo para compreensão do sistema (Halim et al., 2012), e as ferramentas de modelagem SysML, tais como Topcased e Papyrus ainda não possuem um nível bom de maturidade nesse tipo

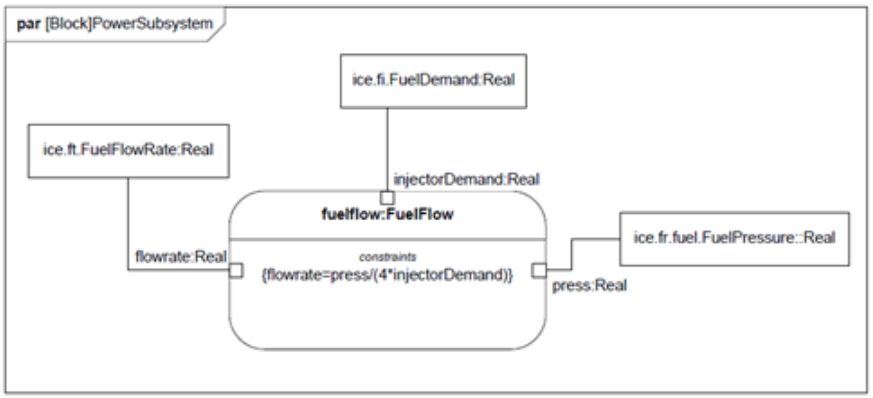

Figura 2.10: Exemplo de diagrama Paramétrico (extraído de OMG (2012))

de modelagem (Novakovic et al., 2013). Um exemplo disso é o fato do suporte definitivo ao diagrama paramétrico na ferramenta Papyrus ter sido incluído apenas no final de março de 2014, na versão 0.10.1v201401061257.

Detalhes da Implementação

Para a inclusão do diagrama Paramétrico foi utilizado o Topcased e o Papyrus. Vale ressaltar que foi utilizada uma versão *beta* do Papyrus, ainda em desenvolvimento, pois as versões definitivas ainda não davam suporte ao diagrama paramétrico. Assim, a versão utilizada do Papyrus foi a 0.10.x.

Como o objetivo é enriquecer a transformação de modelos SysML para Simulink, no contexto da abordagem SyMPLES, a inclusão do diagrama paramétrico seguiu um fluxo baseado na transformação de modelos de Fragal (2013) no seu estado atual. Assim, a inclusão se deu da seguinte forma:

- Identificação dos elementos do diagrama pelo metamodelo: os elementos do metamodelo para o diagrama paramétrico são: *constraint property*, *constraint parameter*, *part*, *reference* e *value*. Todos esses elementos são suportados também pelo editor do Papyrus. Assim, diagramas com esses elementos devem ser transformados para um modelo Simulink correspondente, de acordo com o mapeamento, ilustrado na Tabela - 2.1;

- Implementação da Etapa 1 da transformação: utiliza de regras da linguagem ATL sobre o diagrama paramétrico, e também o metamodelo de entrada SysML e o metamodelo de saída Simulink. Assim, uma representação parcial é produzida, por meio de um arquivo XMI, removendo informações desnecessárias do diagrama

e montando o modelo de saída com os elementos já no formato do metamodelo Simulink utilizado.

- Implementação da Etapa 2: essa implementação conclui o processo, por meio de uma transformação com linguagem de programação Java, utilizando-se do arquivo XMI parcial, bem como de informações do modelo de entrada, para a geração de um *script* Matlab/Simulink. Com a execução desse script no Matlab, o modelo Simulink gráfico é produzido.

Vale ressaltar que a inclusão do diagrama Paramétrico na transformação se deu por meio de um módulo construído como um *plugin*, e seguiu a mesma estrutura da transformação de modelos SysML para Simulink (Fragal, 2013).

O mapeamento dos elementos pode ser visualizado na Tabela - 2.1. Na coluna mais à esquerda, estão os elementos do diagrama paramétrico, todos suportados no editor do Papyrus. Na coluna central, consta uma descrição sucinta do elemento, e na coluna mais à direita, o componente simulink correspondente ao mapeamento.

Tabela 2.1: Mapeamento dos elementos do Diagrama Paramétrico para Simulink

Elemento	**Descrição**	**Componente Simulink**
Constraint Property	Bloco que expressa a restrição em forma de função matemática	Bloco FCN simulink
Constraint Parameter	Parâmetros de outros blocos	Porta IN/OUT
Part	"Instância" de um bloco agregado	Bloco Subsystem
Reference	bloco semelhante ao part mas que não é agregado ao bloco "pai"	Bloco Subsystem
Value	propriedade quantificável com unidades	Bloco Subsystem
Binding Conector	Linha de conexão entre blocos	Conexão Simulink
Comment	Nota de comentários	——
Constraint	Restrição contida em uma nota de comentário	——

Exemplo de Transformação de um Diagrama Paramétrico

Como o modelo SysML avaliado em Fragal (2013) não possui diagrama Paramétrico e para uma avaliação preliminar da inclusão dele na transformação de modelos, utilizou-se de um diagrama paramétrico fictício ilustrado na Figura 2.11. Esse diagrama foi inspirado no contexto dos modelos SysML usados na especificação de um VANT e possui 4 tipos de elementos: blocos do tipo *part*, *constraint property*, *constraint parameter* e também conexões SysML.

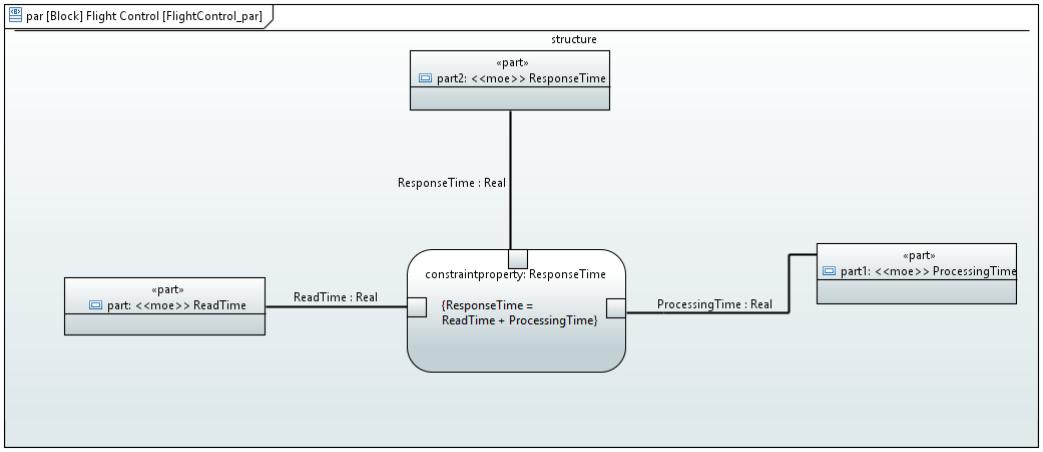

Figura 2.11: Diagrama paramétrico utilizado na transformação

Dessa forma, o diagrama ilustrado na Figura 2.11 foi criado considerando as relações entre o tempo de processamento (*ProcessingTime*) e tempo de leitura (*ReadTime*) de dispositivos, como por exemplo sensores, os quais compõem o tempo de resposta (*ResponseTime* do sistema nomeado *Flight Control*.

O diagrama possui três blocos *part* nomeados como *ReadTime*, *ProcessingTime* e *ResponseTime*, um bloco do tipo *constraint property*, que pode ser considerado como o núcleo do diagrama, pois descreve a relação entre os blocos em uma função matemática, sendo nesse caso uma função de soma de dois elementos. Já os elementos *constraint parameter* são os parâmetros que a função matemática recebe. Esse diagrama também recebe estereótipos SysML para representação de MOEs, para uma suposta avaliação do impacto dessas medidas em requisitos do sistema.

Aplicando-se a primeira etapa da transformação, que é constituída de regras ATL, produz-se um arquivo XMI de saída. As regras são utilizadas para buscar os elementos do diagrama paramétrico e gerar um arquivo XMI com os elementos mapeados ao metamodelo Simulink. Por exemplo, as informações do diagrama paramétrico como as conexões entre os blocos são mantidas no XMI, porém o elemento em si é outro e é

correspondente ao metamodelo Simulink. Nesse caso, esse elemento é do tipo *lines*, o qual representa a conexão entre blocos funcionais, podendo ser visualizado na Figura 2.12. O arquivo XMI completo obtido com a transformação do diagrama da Figura 2.11 pode ser visualizado no Apêndice A deste trabalho.

```
      </children>
    </parts>
    <lines name="ReadTime : Real" type="Connector" source_uml="_YRrJEHllEeO4scp-g95uYQ"
destination_uml="_jQ6TQHlkEeO4scp-g95uYQ"/>
    <lines name="ProcessingTime : Real" type="Connector" source_uml="_W8KHoHllEeO4scp-g95uYQ"
destination_uml="_G5LMMHllEeO4scp-g95uYQ"/>
    <lines name="ResponseTime : Real" type="Connector" source_uml="_XxZtkHllEeO4scp-g95uYQ"
destination_uml="_MMECIHllEeO4scp-g95uYQ"/>
  </sim:Model>
</xmi:XMI>
```

Figura 2.12: Recorte do arquivo XMI de saída produzido pela aplicação da etapa 1 da transformação

Após a aplicação da etapa 1 da transformação, pode-se aplicar a etapa 2 utilizando-se do arquivo XMI (e também o modelo de entrada da etapa 1) como entrada para essa etapa. Essa segunda etapa foi construída por meio da linguagem Java e ela cria um *script* Matlab (conhecido como *M-File*) que, ao ser executado no ambiente Matlab/Simulink, produz o modelo Simulink final.

Os comandos do *script* são necessários para criação dos elementos Simulink, tais como blocos do tipo *Subsystem*. O script produzido com a transformação pode ser visualizado no Apêndice B deste trabalho.

Todos os elementos que são criados respeitam exclusivamente o mapeamento, exceto os blocos *Scope* e *Signal Generator*. Esses blocos foram adicionados implicitamente por uma questão de conveniência: o modelo gerado estará pronto para a simulação no Simulink. Dessa forma, não é necessário que seja realizada a adição de outros blocos para permitir a simulação, apenas pode ser necessário que esses blocos sejam parametrizados (utilização de um intervalo de valores).

Executando-se o script no ambiente Matlab/Simulink, o modelo Simulink final é produzido. Esse modelo final é composto por um bloco, nesse caso nomeado como *Flight Control*, e dentro dele o resultado da transformação do diagrama paramétrico, podendo ser visualizado na Figura 2.13. Na figura pode ser visualizado que cada bloco *part* do diagrama Paramétrico foi transformado em um bloco *Subsystem*, e o bloco *constraint property* transformado em dois blocos: *Mux* e *FCN*. Dentro dos blocos *ReadTime* e *ProcessingTime* no Simulink, foi adicionado o bloco *Signal Generator*, para gerar valores à função especificada no *FCN*, nesse caso uma função de soma. Por outro lado, dentro do bloco *ResponseTime* no Simulink, foi adicionado o bloco *Scope*, para receber o resultado

da função de soma. Assim, em uma simulação, o comportamento poderá ser visualizado por meio de um gráfico produzido com o bloco *Scope*.

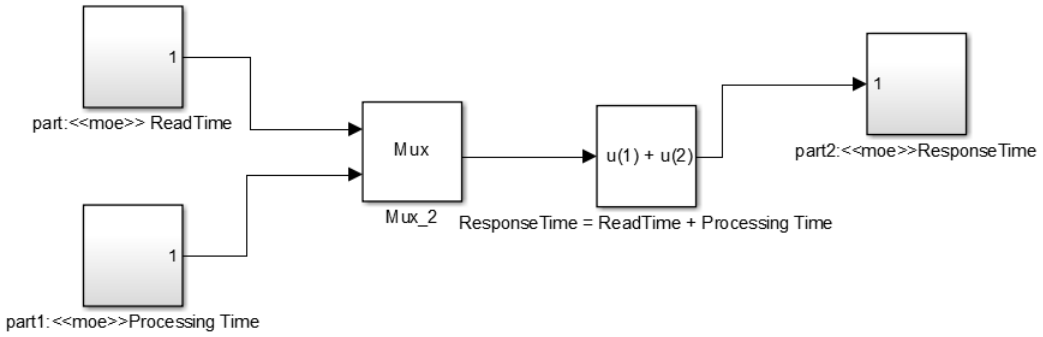

Figura 2.13: Modelo Simulink obtido na transformação

Considerações

No caso da transformação do diagrama paramétrico, a etapa 2 pode ser descrita como a principal etapa da transformação, pois essa etapa identifica a função contida em cada bloco SysML *constraint property* e cria um bloco Simulink FCN.

Função suportada:	Descrição	Forma de uso:
Operadores aritméticos (+,-,*,/,^)	Soma, Subtração, Multiplicação, Divisão e potenciação	W=X+Y-Z
abs, acos, asin, atan	Valor absoluto, Co-seno inverso, seno inverso, tangente inversa	W= nomeF(X)
atan2, ceil, cos, cosh	Arco tangente, arredondamento “para cima”, co-seno, Cosseno hiperbólico	W= nomeF(X)
exp, fabs, floor, hypot	Exponencial, Valor Absoluto (equivalente), arredondamento “para baixo”, “Raiz quadrada da Soma dos quadrados”	W= nomeF(X)
ln, log, log10, pow	logaritmo natural, logaritmo natural (equivalente), logaritmo de base 10, potência	W= nomeF(X)
power, rem, sgn, sin	Potenciação, Resto da divisão, Sinal, Seno	W= nomeF(X)
sinh, sqrt, tan, tanh.	Seno hiperbólico, Raiz quadrada, tangente, tangente hiperbólica	W= nomeF(X)

Figura 2.14: Quadro de funções suportadas pelo bloco FCN do Simulink

Esse bloco FCN implementa um conjunto relativamente grande de funções matemáticas, permitindo a simulação das mesmas. Esse conjunto pode ser visualizado na Figura 2.14 (MathWorks, 2014). Além disso, é nessa etapa que, para cada elemento

do arquivo XMI, é criado um elemento correspondente no *script*, de acordo com o mapeamento.

Com esse módulo, a transformação de modelos SysML para Simulink é enriquecida com o diagrama Paramétrico, permitindo assim a simulação das restrições sobre os valores de propriedades do sistema. Além disso, o bloco FCN e demais blocos possuem suporte à geração de código por meio da ferramenta Simulink Coder (MathWorks, 2014).

2.5 Validação de Transformações de Modelos

A validação de transformação de modelos é um ponto chave para assegurar a qualidade dos modelos transformados (Küster e Abd-El-Razik, 2006). Validação pode ser definida como um conjunto de atividades que garante que o *software* que foi construído é rastreável às exigências do usuário (Pressman, 2010). No contexto das transformações de modelos e neste trabalho, a validação possibilita assegurar que a transformação está de acordo com a sua especificação. As abordagens de validação baseadas em teste de *software* podem ser classificadas em dois tipos (Sen et al., 2009):

- Teste Funcional: é conhecido como teste caixa preta, no qual os modelos de entrada são comparados aos modelos de saída da transformação. É necessária a utilização de modelos de teste que possam detectar erros na transformação.

- Teste Estrutural: é conhecido como teste caixa branca e leva em consideração aspectos internos da transformação de modelos, como por exemplo as regras de transformação.

Para o teste de transformações de modelos, independentemente do tipo, pelo menos três passos devem ser seguidos (Küster e Abd-El-Razik, 2006):

- Geração de Casos de Teste: neste passo, os casos de teste são gerados, de forma automática ou não, de acordo com um dado critério de cobertura. O critério de cobertura utilizado para a geração de casos de teste pode ser de três tipos (Sen et al., 2009): aleatório, partição do domínio de entrada e total. Por meio dos casos de teste é que os testes poderão ser efetivamente executados.

- Criação de um "Oráculo": é necessária a definição de um oráculo, o qual determina o resultado esperado de um teste. As saídas das transformações são comparadas com as entradas para avaliar se o resultado da transformação está de acordo com o esperado e seguindo o mapeamento previamente definido.

- Execução dos testes: nesse passo, os testes são executados, de forma manual ou automatizada.

Com relação à execução dos testes, Tiso et al. (2012) definem duas abordagens para teste de transformações de modelos: teste estático e teste dinâmico. O teste **estático** se refere ao teste de propriedades estáticas dos modelos de saída, como por exemplo, verificar a presença de cadeias de caracteres específicas no modelo de saída. Por outro lado, o teste **dinâmico** se refere ao teste por meio da análise da execução do modelo de saída, caso o mesmo seja compilável ou executável.

2.5.1 Classificação dos tipos de erros em transformações

O processo de análise de resultados a partir da execução dos testes em transformações de modelos também é importante e é diferente da análise para o teste de *software* comum. Uma transformação normalmente se baseia em regras que mapeiam elementos dos modelos de entrada para elementos correspondentes nos modelos de saída. Nesse contexto, uma utilização errada das regras de transformação pode causar os seguintes tipos de erros (Küster e Abd-El-Razik, 2006):

1. **Cobertura ao metamodelo**: as regras de transformação foram implementadas, porém elas não são suficientes para mapear todos os elementos que o metamodelo de entrada permite. Um exemplo é quando uma regra só pode ser aplicada a determinados tipos de elementos.

2. **Modelos sintaticamente incorretos**: quando a regra de transformação causa a geração de um modelo de saída que não está de acordo com as restrições contidas no metamodelo da linguagem de saída.

3. **Modelos semanticamente incorretos**: quando a regra de transformação é aplicada a um modelo de entrada e o modelo de saída produzido é sintaticamente correto, mas não é uma transformação correta do modelo de origem.

4. **Ambiguidade**: a aplicação da regra de transformação permite produzir diferentes saídas a partir do mesmo modelo, tornando a transformação ambígua.

5. **Codificação incorreta**: inclui todos os outros tipos de erros e os de codificação da transformação. Esse tipo de erro é semelhante aos eventuais erros que aparecem na codificação de um *software* comum.

2.5.2 Geração de Casos de Teste

Considerando o teste funcional para validação de transformações de modelos, um caso de teste pode ser considerado como um modelo de entrada à transformação. Nesse contexto, o metamodelo da linguagem de modelagem de entrada pode ser utilizado para geração sistemática de um conjunto de casos de teste (Küster e Abd-El-Razik, 2006).

Um processo de geração de casos de teste baseados em metamodelo (Brottier et al., 2006) pode ser visualizado na Figura 2.15. Esse processo consiste de três passos. O primeiro passo tem o objetivo de criar partições do metamodelo atual, decompondo-o utilizando classes de equivalência (Ostrand e Balcer, 1988). Em seguida, as partições são utilizadas como entrada para criar fragmentos de modelos, os quais são compostos de elementos de determinada partição com valores definidos. Por fim, a partir dos fragmentos, o objetivo do terceiro passo é produzir os casos de teste da transformação, compostos de modelos de entrada.

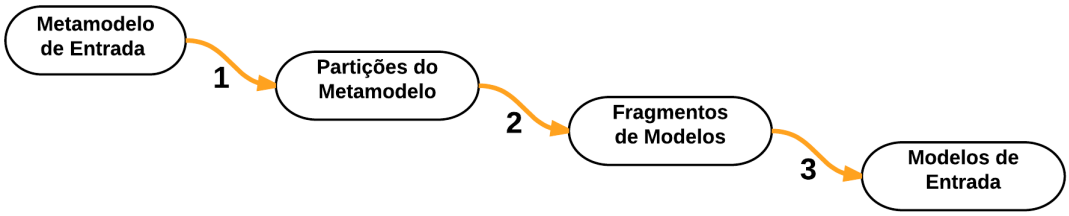

Figura 2.15: Processo de três passos para geração automática de casos de teste (adaptado de Brottier et al. (2006)

Durante essa atividade, dois fatores devem ser levados em conta: **o tamanho de um caso de teste** e **o tamanho do conjunto de casos de teste**. O tamanho de um caso de teste é a quantidade de elementos que o modelo possui, assim um caso de teste pequeno facilita o seu entendimento e permite um diagnóstico mais eficiente quando há a detecção de um erro. Entretanto, diminuir o tamanho de um caso de teste pode implicar em um aumento do tamanho do conjunto. É importante reduzir o conjunto de casos de teste para reduzir o tempo do teste (Fleurey et al., 2004).

2.6 Trabalhos Relacionados

No trabalho de Brottier et al. (2006) foi proposto um algoritmo e uma ferramenta para geração automática de casos de teste. A geração se deu por meio de metamodelos e fragmentos de modelos para teste de transformações. Além disso, foi utilizado o critério de classes de equivalência como critério de cobertura sobre os metamodelos. Uma limitação

encontrada é o fornecimento dos fragmentos de modelos de forma manual, e o escopo do trabalho é reduzido ao processo da geração de casos de teste.

Tiso et al. (2012) fornecem um método de desenvolvimento de transformação de modelos e dois métodos para teste das transformações: uma baseada em teste estático, que consiste em verificar elementos específicos no modelo gerado, e outra baseada em teste dinâmico, que consiste em executar o resultado da transformação, se for código que pode ser compilado e executado. Porém, a concepção de modelos de entrada não é discutida, ou seja, é assumido que o testador já possui (ou os constrói manualmente) os modelos de entrada para teste.

Fleurey et al. (2004) também propõem a utilização de critérios de cobertura sobre o metamodelo utilizado na transformação, e também fazem a comparação de dois algoritmos de geração de casos de teste: um algoritmo que gera os casos de teste de forma sistemática (iterativa), e o segundo utiliza uma abordagem bacteriológica, baseado em algoritmos genéticos. Algumas limitações do trabalho incluem a necessidade de definição de um conjunto inicial de modelos para o algoritmo bacteriológico, e que o escopo do trabalho foi reduzido a avaliar os modelos gerados.

O trabalho proposto por González e Cabot (2012) também se baseia em teste para validação de transformações de modelos. Eles propuseram uma abordagem baseada em teste estrutural, permitindo tratar restrições da transformação que são expressas em OCL (Warmer e Kleppe, 1998). Contudo, a abordagem só permite o teste da transformação se ela foi escrita utilizando a linguagem ATL.

Outra técnica de teste é apresentada em (Guerra, 2012) e em (Lano et al., 2015), os quais se baseiam em teste de transformações de modelos por meio de verificação formal. Essa abordagem de teste é caracterizada como um teste por execução simbólica, ou seja, o teste é realizado sobre os modelos gerados, normalmente máquina de estados.

Guerra et. al. (Guerra et al., 2013) apresentam uma família de linguagens para cobrir todo o ciclo de desenvolvimento de uma transformação de modelos, incluindo o teste. É definida uma linguagem específica para construção de casos de teste, mas também é uma técnica baseada em verificação formal de modelos.

Lin et. al. (Lin et al., 2005) propuseram um *framework* para teste de transformações, fornecendo um ambiente para teste. O ambiente permite checagem de mapeamento entre modelos de entrada/saída de forma automatizada e também a visualização das diferenças entre esses modelos. Contudo, não fornece nenhum mecanismo para encontrar outros tipos de erro. A Tabela - 2.2 mostra um resumo dos principais pontos abordados por cada trabalho comparando-se com esta dissertação.

Tabela 2.2: Resumo dos pontos abordados por cada trabalho

Referência	**Pontos abordados**	**São abordados nesta dissertação?**
(Brottier et al., 2006)	• Geração de Casos de Teste por metamodelos	Sim
(Fleurey et al., 2004)	• Geração de casos de teste por metamodelos • Otimização de algoritmos de geração	Parcialmente
(González e Cabot, 2012)	• Teste estrutural para transformações ATL • Geração de grafo de dependências	Não
(Guerra, 2012)	• Teste dirigido por modelos • Verificação formal	Não
(Lin et al., 2005)	• *Framework* para teste de transformações • Verificação de mapeamento entre modelos de entrada e saída automatizada	Parcialmente
(Tiso et al., 2012)	• Teste estático e dinâmico para transformações	Sim

Wu et al. (2012) apresentam uma revisão sistemática sobre a geração de casos de teste baseados em metamodelos. Juntamente com o trabalho proposto por Küster e Abd-El-Razik (2006), que relata os principais desafios da validação para transformações de modelos, ambos citam que a validação das transformações ainda carece de mais pesquisas para atingir um nível de maturidade maior.

A geração de casos de teste por meio de um modelo de características de uma LPS será utilizada neste trabalho e será comparada com a geração baseada em metamodelo. No primeiro caso uma LPS será utilizada para o teste da transformação com modelos de um domínio específico. A geração pelo metamodelo por outro lado será baseada no metamodelo linguagem SysML, obtendo-se assim modelos genéricos para o teste da transformação da SyMPLES. As técnicas utilizadas neste trabalho são especificadas em uma abordagem de validação para transformações de modelos complexas.

2.7 Considerações Finais

Este capítulo apresentou os conceitos principais para o desenvolvimento deste trabalho de mestrado, incluindo tópicos como MDE, LPS, abordagem SyMPLES e validação de transformação de modelos. Este trabalho utiliza os conceitos de validação por meio de teste para aplicar sobre a transformação de modelos da abordagem SyMPLES.

3

A Abordagem VAMT

3.1 Introdução

Este capítulo apresenta uma abordagem baseada em teste funcional para validação de transformações de modelos complexas, nomeada como VAMT: *Validation Approach for Model Transformations* (Abordagem de Validação para Transformações de Modelos). A abordagem é composta de três atividades principais para a validação: **Geração de casos de teste**, **Execução dos testes** e **Análise de resultados**. A primeira atividade pode ser realizada de duas formas nesta abordagem, utilizando um modelo de características de uma LPS, ou utilizando um metamodelo da linguagem SysML. A segunda atividade consiste basicamente na execução da transformação e a última analisa os resultados da transformação para identificar erros.

A principal justificativa pelo uso de teste funcional na VAMT é devido às diferentes linguagens que podem ser usadas em cada etapa de uma transformação, dificultando o uso de teste estrutural. A transformação da SyMPLES é um exemplo. A especificação da abordagem de validação é voltada para quaisquer transformações de modelos que sejam constituídas por etapas de transformação, no entanto, as implementações realizadas são voltadas para a transformação da abordagem SyMPLES. A aplicação e avaliação desta abordagem serão apresentadas no Capítulo 4.

3.2 Atividades da Abordagem de Validação Proposta

As três atividades principais que compõem o processo da abordagem VAMT são detalhadas na Figura 3.1. Elas são representadas pelas áreas numeradas 1, 2 e 3 e determinam a sequência do processo.

Cada retângulo com bordas arredondadas representa um passo para realização de cada atividade da validação. A primeira atividade é a **Geração de Casos de Teste**, e utiliza como entrada um critério de cobertura definido. De acordo com um critério de cobertura que pode ser total, particionado ou aleatório, os casos de teste são gerados para entrada da **Etapa 2.1** da transformação. A abordagem proposta inclui dois tipos de geração de casos de teste:

- **Geração de Casos de Teste por meio do Metamodelo**: utiliza o metamodelo da linguagem SysML para gerar modelos de entrada da transformação. Um critério de cobertura é aplicado sobre o metamodelo de entrada que define o quanto deste metamodelo será testado.

- **Geração de Casos de Teste por meio de LPS**: utiliza uma LPS qualquer para gerar um conjunto de modelos SysML de entrada, desde que ela seja especificada por meio da abordagem SyMPLES e que seja fornecido o seu modelo de características. Nesse caso, o critério de cobertura é aplicado sobre o modelo de características da LPS para determinar a quantidade de modelos a serem gerados.

A atividade **Execução dos Testes** é constituída pelos passos Execução, Coleta de Resultados, Verificação de Mapeamento e Teste Dinâmico, mostrados na Figura 3.1. O teste nada mais é do que a execução de uma etapa da transformação utilizando modelos como entrada. No caso da transformação da SyMPLES, a execução da **Etapa 2.1** na Figura 3.1 utiliza modelos SysML como entrada e os transforma em uma representação intermediária, caso não ocorra nenhum erro durante a execução. No teste de cada etapa os resultados são coletados, os quais definem se houveram erros na transformação. A execução dos testes em cada etapa da transformação produzirá um ou mais modelos intermediários ou de saída. No caso da transformação da SyMPLES, os modelos produzidos pela execução da **Etapa 2.1** são utilizados para o teste da **Etapa 2.2** da transformação. O ciclo entre as atividades de Execução e de Coleta de Resultados termina após o teste da última etapa da transformação.

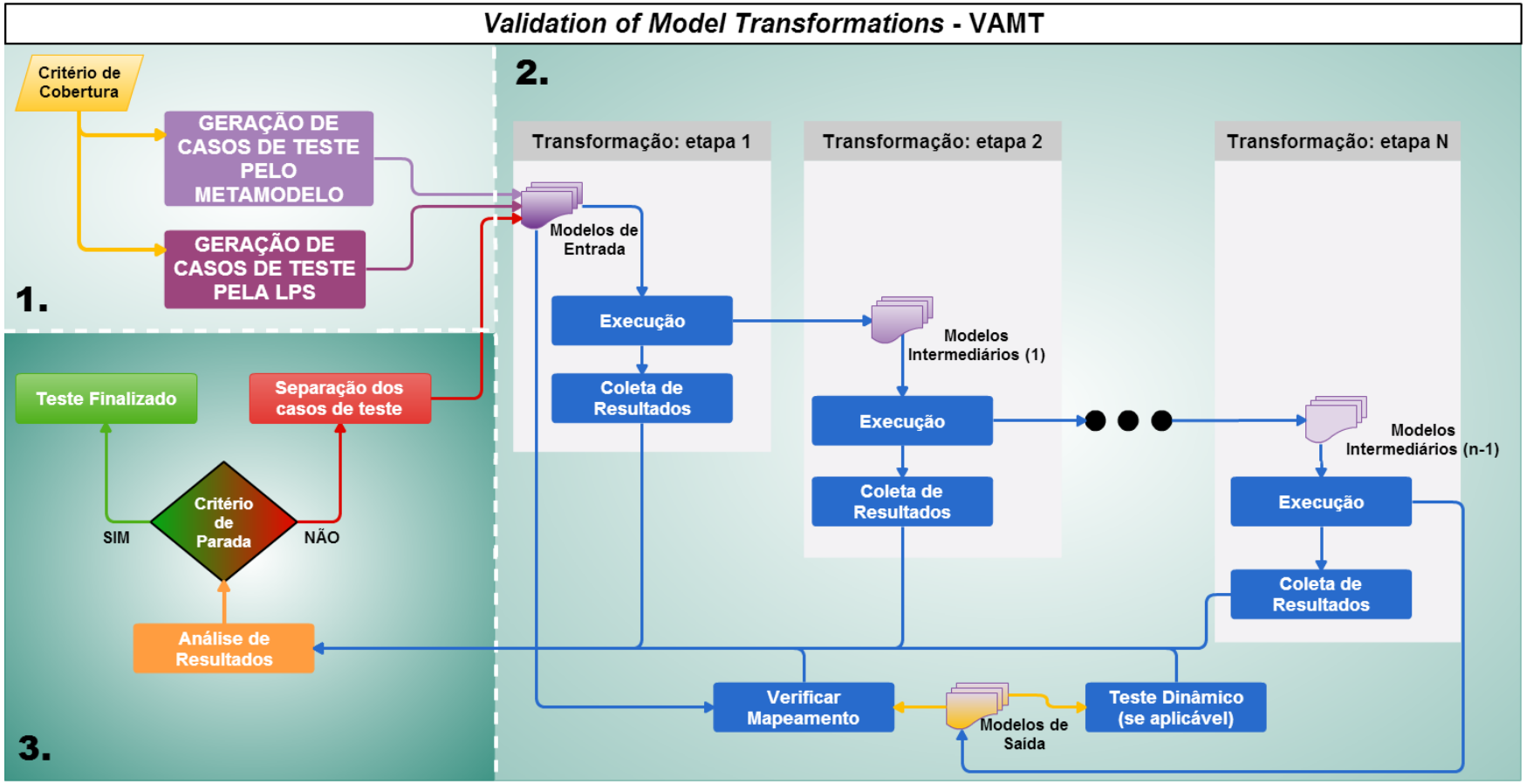

Figura 3.1: Abordagem de Validação Proposta

Um potencial problema que pode surgir na atividade de **Execução dos Testes** é se existirem erros na etapa anterior da transformação que impeçam a geração de um modelo de saída. Uma solução para isto é que o teste de cada etapa seja realizado em sequência. Outra possível solução é a separação da atividade de **Geração de Casos de Teste** para cada etapa da transformação, porém, nem sempre os metamodelos intermediários são bem definidos, e, além disso, uma alteração em um metamodelo pode demandar mudança nos demais.

Outra forma de execução dos testes para a validação da transformação é o teste dinâmico (Tiso et al., 2012). Ele só é aplicável se os modelos de saída produzidos pela transformação são códigos compiláveis/executáveis. De posse dos modelos de saída, é possível testar sua execução para encontrar erros.

Após a execução da última etapa da transformação, é necessário iniciar a Verificação de Mapeamento para garantir que o modelo transformado possui os elementos esperados, de acordo com o mapeamento. Por exemplo, se um diagrama paramétrico possui um bloco do tipo *part*, de acordo com o mapeamento, no modelo de saída deverá aparecer o elemento correspondente ao bloco *part*.

A atividade **Análise de Resultados** tem bastante importância na validação, e se baseia nos dados coletados com a execução dos testes como os erros na transformação e também nos resultados das verificações e testes sobre os modelos gerados. A análise é constituída também pela separação de casos de teste que mostraram erros dos casos de teste que não mostraram erros. Os casos de teste que mostraram erros deverão ser executados novamente, após a correção dos erros na transformação. Por fim, o critério de parada é usado para decidir se a transformação está suficientemente validada ou não. Quando o critério de cobertura utilizado não mostrou erros e foi julgado suficiente para a validação, ou se o critério de cobertura total foi usado e não mostrou erros, o teste pode ser finalizado. A seguir, exemplos de modelos de entrada e de saída são detalhados e também os tipos de geração de casos de teste implementados neste trabalho. É importante ressaltar que a abordagem proposta é genérica mas as implementações são voltadas à validação da transformação SysML para Simulink.

3.2.1 Modelos de Entrada

A transformação SysML para Simulink pode ser classificada como uma transformação *Model-to-Model*, conforme apresentado na Seção 2.3. Os modelos de entrada para a transformação podem ser quaisquer modelos escritos em linguagem SysML que respeitem algumas restrições impostas na transformação, as quais serão apresentadas na subseção

3.2.4. Um exemplo de restrição é que apenas serão transformados os modelos que contenham pelo menos um dos seguintes diagramas: Definição de Blocos, Interno de Bloco, Paramétrico ou Máquina de Estados.

Um exemplo de modelo de entrada é apresentado na Figura 3.2. Ele é composto apenas por um diagrama Interno de Bloco, adaptado de Fragal (2013). A transformação recebe esse modelo de entrada e o transformará de acordo com os elementos que estão contidos no modelo. Nessa figura, o bloco *Yapa2* possui o estereótipo "subsystem" e portanto esse elemento será transformado para um bloco Simulink correspondente.

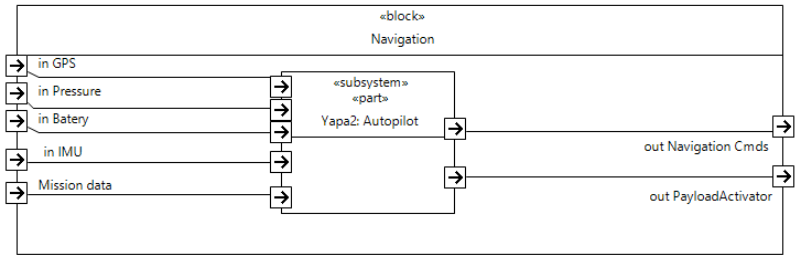

Figura 3.2: Exemplo de modelo de entrada para a transformação SysML para Simulink (adaptado de Fragal (2013)).

3.2.2 Modelos de Saída

Os modelos de saída produzidos pela transformação SyMPLES são modelos Simulink em formato de *script* Matlab. Um exemplo de *script* Matlab que possui elementos Simulink é apresentado no Apêndice A. Cada *script* é produzido de acordo com um modelo de entrada e pode ser executado no ambiente Matlab, resultando na visualização do modelo Simulink.

Um exemplo do modelo de saída após a sua execução no Matlab é apresentado na Figura 3.3, adaptada de Fragal (2013). Ela mostra o modelo Simulink correspondente à transformação do modelo de entrada apresentado na subseção 3.2.1. É possível notar que o bloco SysML *Yapa2* foi transformado em um bloco Simulink do tipo *subsystem*.

3.2.3 Geração dos casos de teste por meio de uma LPS

A geração de casos de teste por meio de LPS foi implementada, pois uma LPS é usada para gerar vários produtos configurados por meio da abordagem SyMPLES e estes produtos são modelos SysML de entrada para a transformação. Assim, é possível usá-los para o teste da transformação de modelos.

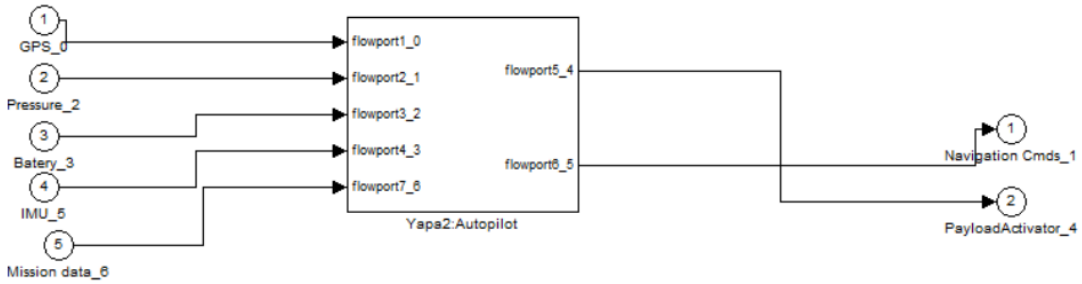

Figura 3.3: Exemplo de modelo de saída produzido pela transformação (adaptado de Fragal (2013)).

A atividade de Geração dos Casos de Teste utiliza como entrada o modelo de caracterísiticas de uma LPS para gerar modelos de entrada para a primeira etapa de transformação. A geração também utiliza como entrada um critério de cobertura sobre os possíveis produtos da LPS, o qual deve respeitar as seguintes propriedades:

- Quando o modelo de características da LPS for pequeno em termos da quantidade de possíveis produtos a serem gerados, o critério de cobertura aleatório, particionado ou total poderá ser utilizado, conforme descrição no Capítulo 2.

- Quando o modelo de características for médio ou grande, o critério de cobertura deve ser o particionado. Nesse caso, a LPS pode ser configurada em um número muito grande de produtos diferentes (Oster et al., 2011). Essa implementação utiliza para o cálculo da quantidade de produtos um BDD cuja estrutura é intensiva em memória (Benavides et al., 2007). O tamanho desta estrutura pode exceder as capacidades de memória disponíveis para a implementação do gerador.

Detalhes da implementação

Na Figura 3.4 pode ser visualizado o processo de geração de casos de teste baseado em LPS. A implementação do gerador de casos de teste utiliza a ferramenta pure::variants para especificação da LPS e dos produtos configurados. Após gerar e configurar cada *Variant Descripton Model* (VDM), os produtos devem ser instanciados e assim os modelos de entrada são gerados para o teste.

Cada VDM é gerado por meio da análise automatizada do MC em formato XML que deve ser inserido no gerador. Dessa forma, o gerador cria um VDM para cada configuração possível dos produtos da LPS. No entanto, se o critério de cobertura não for total, nem todos os VDM serão gerados, apenas os definidos pelo critério. A Figura 3.5 mostra um recorte do código da implementação desse gerador, mostrando como o critério é aplicado

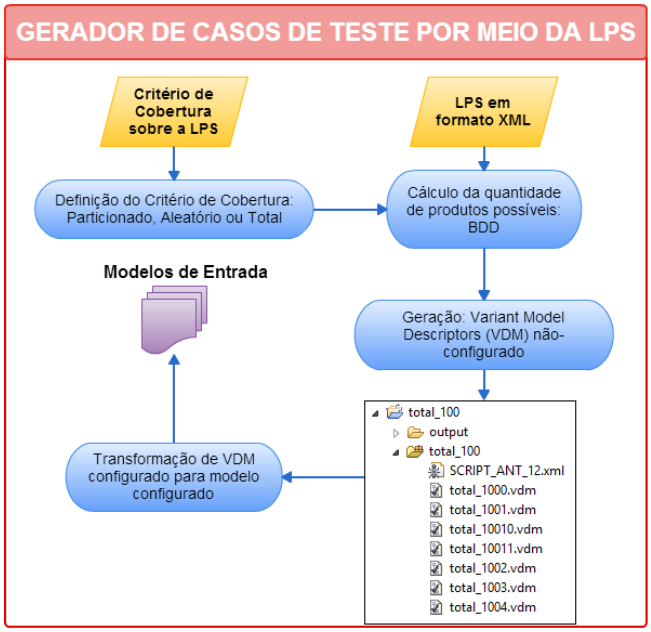

Figura 3.4: Representação do gerador de casos de teste baseado em LPS.

sobre o resultado da análise automatizada. Para isso, foi utilizado um conjunto de classes Java baseado na análise via diagrama BDD, disponível na ferramenta SPLOT (Mendonca et al., 2009), cujo código é aberto.

Após o cálculo das possíveis configurações da LPS, a quantidade de modelos que serão gerados é calculada, considerando o critério de cobertura. A variável "criterio" no recorte de código recebe o resultado desse cálculo. O percentual de cobertura é informado pelo usuário. Vale ressaltar também que a implementação do gerador realizada neste trabalho também utilizou como base a implementação do *plugin SysML Importer* (Silva et al., 2013).

Para a implementação desse gerador foi utilizada a linguagem Java, o conjunto de classes disponíveis para utilização de um BDD por meio da ferramenta SPLOT, e também o código do *SysML Importer*, para criação de cada VDM.

Uma limitação da implementação do gerador de casos de teste é que a escolha das características dos produtos deve ser manual. Após a escolha, executa-se um *script* ANT para transformação automática dos VDMs configurados para modelos configurados, instanciando-os automaticamente. Esse *script* é gerado automaticamente no momento da geração dos casos de teste.

```
// ////////////////////////////////////////////////////////////////////////////
//      BUSCA DAS CONFIGURACOES POSSIVEIS
BDDReasoningExample bdd = new BDDReasoningExample();
double p =bdd.retornarConfsValidasPL();
if (p > 99999){ //too much
    JOptionPane.showMessageDialog(null, "ERRO: Muitas possibilidades de produtos.");
    return false;
}
/*
 *        CRITERIO DE COBERTURA INFORMADO:
 */
double C = Double.parseDouble(m_TargetPage.getCRITERIO());
C = Math.round(C);

//Critério de cobertura definido: PERCENTUAL
double criterio = (C * p)/100;

System.out.println("VDMs: "+criterio);
// ////////////////////////////////////////////////////////////////////////////
```

Figura 3.5: Recorte do código da implementação do gerador de casos de teste por meio da LPS.

3.2.4 Geração de casos de teste por meio do Metamodelo

O gerador de casos de teste baseado em Metamodelo foi implementado como um *plugin* para a plataforma Eclipse, considerando o metamodelo da linguagem SysML e o perfil de blocos funcionais da abordagem SyMPLES. Dessa forma, os casos de teste são gerados extraindo-se os possíveis elementos do metamodelo e combinando-os com os estereótipos da abordagem. Nesse caso há três requisitos para a geração de casos de teste por meio do metamodelo:

- Definir o metamodelo para a geração dos casos de teste: é necessário escolher quais elementos, diagramas e modelos deverão ser gerados. Para este caso, o metamodelo SysML é usado na geração dos casos de teste, e o escopo da implementação foi reduzido para os diagramas estruturais (Definição de Blocos, Interno de Bloco e Paramétrico).

- Definir critérios de cobertura: de forma similar à geração baseada em LPS, os critérios de cobertura usados foram: aleatório, parcial e total.

- Escolher uma política de geração: deve-se definir pelo menos uma organização dos elementos nos modelos gerados (Fleurey et al., 2004). Na avaliação do Capítulo 4, as duas formas citadas abaixo foram comparadas:

  – Um conjunto de elementos para o mesmo diagrama **(N-para-1)**. Elementos de tipos diferentes são separados e o conjunto é limitado em 5 elementos de mesmo tipo no máximo.

– Um diagrama para cada novo elemento **(1-para-1)**.

Dificilmente todos os possíveis elementos de um metamodelo são usados por uma transformação (Fleurey et al., 2004). Dessa forma, uma proposta baseada em cobertura ao metamodelo pode gerar inúmeros casos de teste inúteis. É necessário determinar o que é relevante para a transformação.

O mapeamento dos elementos SysML para Simulink da transformação foi analisado para descobrir um percentual de utilização do metamodelo SysML. A Tabela - 3.1 mostra a quantidade de elementos que podem ser utilizados em relação ao total. Por exemplo, o metamodelo SysML indica que podem ser usados 38 elementos diferentes em diagramas de Definição de Blocos, mas foi constatado que apenas 3 desses elementos são mapeados por regras de transformação. Assim, as regras mapeiam aproximadamente 8% dos elementos em relação ao total. Na análise foi considerado o metamodelo SysML usado no *Papyrus* versão 0.8.2v201204301641 e a versão 0.10.1v201401061257 para a análise do diagrama Paramétrico.

Tabela 3.1: Elementos do metamodelo que são efetivamente utilizados pela transformação SysML para Simulink

Diagrama	**Elementos possíveis**	**Percentual de elementos que a transformação utiliza**
Definição de Blocos	38	8%
Interno de Bloco	9	56%
Paramétrico	8	75%
Máquina de Estados	13	100%
Média	**-**	**60%**

Nesse sentido, uma utilização de um gerador pelo metamodelo poderia gerar casos de teste que não são interessantes à transformação de modelos e, portanto, podem ser descartados.

Uma análise sobre o metamodelo SysML também foi realizada para identificar quantos casos de teste poderiam ser gerados, considerando os relacionamentos entre elementos dos diagramas. A Tabela - 3.2 mostra a análise realizada para os diagramas: Definição de Blocos, Interno de Bloco, Máquina de Estados e Paramétrico. A análise se baseou no princípio de que cada conexão entre dois elementos, seja ela de qualquer tipo: composição, dependência, entre outras, e permitindo a repetição que deveria ser testada separadamente. Dessa forma, os elementos foram organizados em arranjos tomados dois a dois com repetição, para cada diagrama que a transformação considera.

Em alguns casos particulares, foi realizada a contagem das possibilidades ao invés dos arranjos. Isso porque nem sempre um elemento pode ser conectado usando um tipo específico de associação. Por exemplo, no diagrama de Definição de Blocos, apenas alguns elementos e em determinadas ordens podem ser conectados com *Interface Realization*.

Tabela 3.2: Análise das quantidades totais de casos de teste considerando arranjos de elementos do metamodelo SysML.

Diagrama	**Tipo Relacionamento**	**Forma de cálculo**	**Quantidade de casos de teste**
Definição de Blocos	*Association*	$AR(9,2) = 9^2$	81
	Directed Association	$AR(9,2) = 9^2$	81
	Composition	$AR(9,2) = 9^2$	81
	Directed Composition	$AR(9,2) = 9^2$	81
	Agregation	$AR(9,2) = 9^2$	81
	Directed Agregation	$AR(9,2) = 9^2$	81
	Dependency	$AR(16,2) = 16^2$	256
	Generalization	$AR(10,2) = 10^2$	100
	Interface Realization	Contagem	6
	Usage	$AR(16,2) = 16^2$	256
	Subtotal	-	**1104**
Interno de Bloco	*Connector*	$AR(7,2) = 7^2$	49
	Dependency	$AR(7,2) = 7^2$	49
	Subtotal	-	**98**
Máquina de Estados	*Transition*	$AR(10,2) = 10^2 + 20$	120
	Subtotal	-	**120**
Paramétrico	*Connector*	Contagem	4
	Dependency	$AR(5,2) = 5^2$	25
	Subtotal	-	**29**
-	**TOTAL**	-	**1351**

Somando-se as possibilidades de arranjos (AR) mostrados na Tabela - 3.2, tem-se o total de casos de teste necessários. Foi constatado que considerando apenas os elementos

tomados dois a dois com repetição, salvo casos especiais, o total máximo de casos de teste é igual a **1351**. Esse número elevado de casos de teste pode inviabilizar a execução do teste, principalmente quando não há recursos de automatização disponíveis para o testador.

Além disso, há restrições embutidas nas transformações para que a mesma seja usada de forma correta. Na transformação SyMPLES, as restrições são listadas abaixo:

- Diagramas de Definição de Bloco e Interno de Bloco requerem o uso dos estereótipos SyMPLES. Elementos sem estereótipos não são transformados.
- É necessário o uso de portas para conectar elementos.
- Portas do tipo inout não foram mapeadas e, portanto, não são consideradas na transformação.
- Portas que recebam mais que uma ligação não são transformadas.
- O diagrama Paramétrico requer o uso de pelo menos uma função matemática embutida em um elemento do tipo *Constraint Property*.

Dessa forma, o metamodelo SysML foi reduzido para atender às restrições da transformação. Isso significa que só serão gerados modelos que atendam aos requisitos da transformação. Vale ressaltar também que o gerador considerou apenas os diagramas estruturais SysML: Definição de Blocos, Interno de Bloco e Paramétrico.

O metamodelo reduzido foi combinado com o perfil de blocos funcionais da abordagem SyMPLES. Isso porque a primeira restrição da transformação requer o uso deles para uma transformação correta. Os estereótipos utilizados na implementação são mostrados na Figura 3.6, obtidos por meio da implementação atual do perfil de blocos funcionais SyMPLES no editor Papyrus.

São 48 estereótipos da abordagem SyMPLES, e cada um corresponde a um bloco Simulink diferente. Assim, o gerador de casos de teste deve considerar cada um dos estereótipos na geração.

Detalhes da implementação

A Figura 3.7 mostra o pseudocódigo do gerador de casos de teste baseado no metamodelo. A política de geração e o critério de cobertura são usados para determinar a organização de elementos nos modelos e a quantidade de modelos gerados.

A política de geração N-para-1 se baseia no tipo de elemento. Cada modelo terá no máximo 5 elementos distintos, mas do mesmo tipo. Por exemplo: 5 blocos *part* poderão

Stereotype	Information	Stereotype	Information
Absolute	SyMPLES-ProfileFB_internal::Absolute	MultiportSwitch	SyMPLES-ProfileFB_internal::MultiportSwitch
BusCreator	SyMPLES-ProfileFB_internal::BusCreator	Mux	SyMPLES-ProfileFB_internal::Mux
BusSelector	SyMPLES-ProfileFB_internal::BusSelector	Product	SyMPLES-ProfileFB_internal::Product
Constant	SyMPLES-ProfileFB_internal::Constant	Ramp	SyMPLES-ProfileFB_internal::Ramp
DataTypeConversion	SyMPLES-ProfileFB_internal::DataTypeConversio	RandomNumber	SyMPLES-ProfileFB_internal::RandomNumber
Delay	SyMPLES-ProfileFB_internal::Delay	RateTransition	SyMPLES-ProfileFB_internal::RateTransition
Demux	SyMPLES-ProfileFB_internal::Demux	RepeatingSequence	SyMPLES-ProfileFB_internal::RepeatingSequence
Display	SyMPLES-ProfileFB_internal::Display	Scope	SyMPLES-ProfileFB_internal::Scope
EnabledSubsystem	SyMPLES-ProfileFB_internal::EnabledSubsystem	SignalGenerator	SyMPLES-ProfileFB_internal::SignalGenerator
FloatingScope	SyMPLES-ProfileFB_internal::FloatingScope	SignalSpecification	SyMPLES-ProfileFB_internal::SignalSpecification
Function	SyMPLES-ProfileFB_internal::Function	SineWave	SyMPLES-ProfileFB_internal::SineWave
FunctionBuilder	SyMPLES-ProfileFB_internal::FunctionBuilder	Step	SyMPLES-ProfileFB_internal::Step
Gain	SyMPLES-ProfileFB_internal::Gain	StopSimulation	SyMPLES-ProfileFB_internal::StopSimulation
If	SyMPLES-ProfileFB_internal::If	Subsystem	SyMPLES-ProfileFB_internal::Subsystem
IMU	SyMPLES-ProfileFB_internal::IMU	Subtract	SyMPLES-ProfileFB_internal::Subtract
Integrator	SyMPLES-ProfileFB_internal::Integrator	Sum	SyMPLES-ProfileFB_internal::Sum
Joystick	SyMPLES-ProfileFB_internal::Joystick	Switch	SyMPLES-ProfileFB_internal::Switch
KalmanFilter	SyMPLES-ProfileFB_internal::KalmanFilter	SwitchCase	SyMPLES-ProfileFB_internal::SwitchCase
ManualSwitch	SyMPLES-ProfileFB_internal::ManualSwitch	TCP/IPReceive	SyMPLES-ProfileFB_internal::TCP/IPReceive
Mean	SyMPLES-ProfileFB_internal::Mean	TCP/IPSend	SyMPLES-ProfileFB_internal::TCP/IPSend
Memory	SyMPLES-ProfileFB_internal::Memory	TriggeredSubsystem	SyMPLES-ProfileFB_internal::TriggeredSubsysten
MinMax	SyMPLES-ProfileFB_internal::MinMax	VideoDisplay	SyMPLES-ProfileFB_internal::VideoDisplay
MinMaxRunningRese...	SyMPLES-ProfileFB_internal::MinMaxRunningRes	VideoInput	SyMPLES-ProfileFB_internal::VideoInput
MT3329:GPS	SyMPLES-ProfileFB_internal::MT3329:GPS	XYGraph	SyMPLES-ProfileFB_internal::XYGraph

Figura 3.6: Estereótipos da abordagem SyMPLES utilizados na implementação do gerador baseado no metamodelo.

compor um caso de teste, mas 3 blocos *part* e 2 blocos *reference* deverão estar em casos de teste separados.

Para a implementação a linguagem Java foi utilizada. A geração dos modelos de entrada foi realizada por meio do pacote Java *Document Object Model* (DOM) (Oracle, 2015). Esse pacote possibilita a manipulação dos arquivos em formato XML que compõem cada modelo.

3.2.5 Verificação de Mapeamento

Um problema ao se automatizar a geração de casos de teste é que os modelos gerados podem ser difíceis de se interpretar por um testador humano, dependendo da política usada (Baudry et al., 2006). Isso pode dificultar a localização dos erros, pois se um caso de teste mostra um erro, o testador deve entender o modelo para descobrir qual parte da transformação foi executada ao utilizar o dado modelo de entrada. Assim, o testador pode identificar onde está o erro na transformação.

Além disso, quando a geração de casos de teste produzir muitos modelos de entrada, e consequentemente obtiver muitos modelos de saída com a execução dos testes, a verificação do mapeamento para cada modelo de entrada/saída pode se tornar tediosa e propensa

```
1. Ler metamodelo;
2. Estabelecer política de geração P:
3.    se P é 1-para-1 então
4.          para cada elemento dentro do critério de cobertura,
5.                copie o projeto em branco para o destino
6.                insira o elemento
7.                salve o modelo
8.    se P é N-para-1 então
9.          para cada conjunto de elementos dentro do critério de cobertura
10.               copie o projeto em branco para o destino
11.               insira o conjunto
12.               salve o modelo
```

Figura 3.7: Pseudocódigo da implementação do gerador de casos de teste baseado no Metamodelo.

a erros. Dessa forma, é necessário automatizar também a verificação, para facilitar a identificação de erros de mapeamento (tipo 3 da classificação apresentada na Seção 2.5).

Para automatizar a Verificação de Mapeamento, conforme a abordagem proposta, foi desenvolvido um programa Java, aqui chamado de **Oráculo**. Essa implementação independe do tipo de geração de casos de teste utilizada. Ela consiste em verificar o mapeamento de elementos na transformação: se um elemento aparece no modelo de entrada, o seu correspondente transformado deve aparecer no modelo de saída, caso contrário, a transformação não pode ser validada, pois um erro foi encontrado. A Figura 3.8 mostra a necessidade dos modelos de entrada e dos modelos de saída para a realização da verificação de mapeamento. Os modelos de entrada são obtidos após a geração dos casos de teste e os de saída após a execução da transformação.

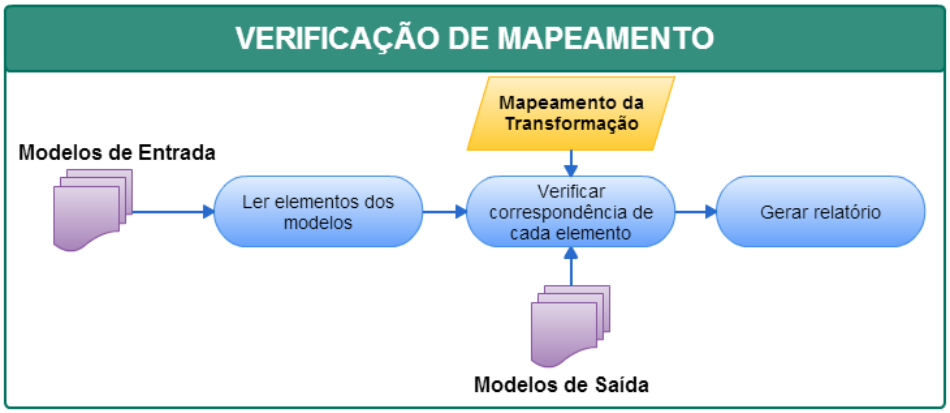

Figura 3.8: Esquemático da Verificação de Mapeamento.

É necessário também o conhecimento do mapeamento utilizado na transformação, para que se possa definir qual o resultado esperado dela. Um exemplo de mapeamento que foi

utilizado na implementação do Oráculo pode ser visualizado na Tabela - 2.1. Essa tabela mostra o mapeamento dos elementos do diagrama Paramétrico para elementos Simulink.

Na Figura 3.9 é mostrado o pseudocódigo da implementação do Oráculo. Destaca-se que nessa implementação foi considerada apenas a verificação no destino. Isso significa que todos os elementos de entrada foram pesquisados pelo correspondente na saída, mas não o contrário. Caso a transformação adicione outros elementos na saída, eles não serão considerados nessa implementação.

```
1.    Para cada modelo de entrada:
2.          Abrir o modelo e obter os elementos contidos;
3.          Para cada elemento do modelo de entrada:
4.                Verificar a correspondência no modelo de saída;
5.                Se o elemento correspondente não for encontrado
6.                      Adicionar uma entrada no relatório de erros.
7.                Se for encontrado, pular para próximo elemento.
8.    Armazenar o log.
```

Figura 3.9: Pseudocódigo da implementação do Oráculo.

Vale ressaltar que a implementação apenas verifica elementos que possuam mapeamento especificado. Além disso, por simplicidade, portas e conexões entre elementos não são verificadas automaticamente, apenas são considerados os tipos de blocos válidos para a transformação. A verificação dos relacionamentos entre os elementos pode ser realizada facilmente de forma visual. Dessa forma, a verificação do mapeamento também atendeu às restrições da transformação da mesma forma que a geração pelo metamodelo.

3.2.6 Teste Dinâmico

Pelo fato da transformação SysML para Simulink produzir como modelos de saída *scripts* Matlab executáveis, o teste sobre eles é importante para a validação da transformação (Tiso et al., 2012). Se os modelos não executarem corretamente, pode-se concluir que a transformação não foi capaz de gerá-los conforme esperado.

Nesse sentido, o teste dinâmico foi incluído na abordagem para fornecer maior nível de validação. A abordagem abrange tanto o teste estático, por meio da execução da transformação e verificação do mapeamento, quanto o teste dinâmico sobre os modelos de saída. Vale ressaltar que este último só pode ser aplicado se os modelos de saída forem compiláveis ou executáveis.

Para o caso da transformação da abordagem SyMPLES, o teste dinâmico é realizado de forma automatizada por meio de um *script* Matlab auxiliar. Um trecho desse *script* pode ser visualizado na Figura 3.10.

```
diary('logDiaryIBDp1.txt');
fid = fopen('logTesteDinamicoIBDp1.txt','wt');

countErros =0;
d = dir([myDir filesep '*.m']);
for jj=1:numel(d)
    [~,name,ext] = fileparts(d(jj).name);
    try
        %execução do modelo de saída
        feval(name);
    catch EXCEPT
        lin = EXCEPT.stack.line;
        countErros = countErros+1;
        fprintf(fid,'--- ERRO ENCONTRADO!! ---');
        fprintf(fid,'\nERRO %d: \n Modelo de saída:%s\n Mensagem de erro:%s\n Linha: %d\n',
                countErros, name, EXCEPT.message , lin);
    end
end
fclose(fid);
```

Figura 3.10: Trecho do *script* auxiliar para realização do teste dinâmico.

O *script* realiza a execução de cada modelo de saída da transformação da SyMPLES, por meio do comando *feval*. Caso ocorra algum erro na execução do modelo, ele é armazenado no arquivo *logTesteDinamico*, cujo nome varia entre os diagramas ou políticas avaliadas. Eventuais avisos (*warnings*) são armazenados no arquivo *Diary*. É necessário apenas que os modelos de saída a serem testados se encontrem em um mesmo diretório.

O arquivo de *log* para armazenar erros na execução dos modelos é composto pelas seguintes informações: número do erro no *log*; nome do modelo de saída; mensagem de erro; e o número da linha no modelo que ocasionou o erro. Essas informações auxiliam a identificação da causa do erro na execução do modelo de saída. No entanto relacionar esse erro encontrado no teste dinâmico com a transformação de modelos é difícil e é necessário conhecimento da implementação da transformação.

3.3 Comentários Finais

Alguns comentários e observações foram elaborados para orientar a aplicação da abordagem de validação. São eles:

- **Automatização da Geração de Casos de Teste**: é necessário utilizar uma ferramenta para gerar os casos de teste automaticamente, para reduzir o tempo

gasto nesse processo. Nesse trabalho foram implementados dois geradores de casos de teste voltados à transformação da SyMPLES, mas para outras transformações são necessários outros geradores.

- **Automatização da Execução dos Testes**: nesta abordagem nenhuma ferramenta de automatização da execução dos testes foi implementada, mas para reduzir o tempo gasto na realização do teste, é necessário que a execução seja realizada em um ambiente que suporte a automatização do teste. Por exemplo, para automatizar um teste de uma transformação escrita em ATL, existem 3 ferramentas disponíveis: EUnit (García-Domínguez et al., 2011), ferramenta de teste que pode ser utilizada para o teste de transformações ATL; ANT (Apache, 2014), ferramenta mais genérica, composta de um ou mais scripts de configuração e execução de teste; ou executar a transformação por meio de um programa escrito em linguagem Java.

- **Verificação de Entrada/Saída Automatizada**: é necessário verificar se o mapeamento proposto pela transformação está correto, e verificar cada par (modelo de entrada, modelo de saída) manualmente pode ser muito custoso em termos de tempo, dependendo das características de cada modelo. Assim, recomenda-se que essa etapa seja automatizada.

- **Realização dos testes**: se houverem erros em uma etapa anterior da transformação, os modelos intermediários podem não ser gerados e assim o teste da etapa seguinte pode ficar prejudicado. Dessa forma, recomenda-se que a validação ocorra de forma sequencial entre as etapas ou que a geração dos casos de teste aconteça em cada etapa da transformação.

- **Escolha do Critério de Cobertura**: a escolha do critério de cobertura causa um impacto direto na geração de casos de teste. Deve-se conhecer o domínio de entrada das transformações para não aumentar significativamente a quantidade de casos de teste e, consequentemente, o tempo do teste.

4

Aplicação da Abordagem de Validação na Transformação SysML para Simulink

4.1 Introdução

Nesta seção, serão apresentados os resultados obtidos com a aplicação da abordagem VAMT na transformação de modelos SysML para Simulink, e também um comparativo entre os tipos de gerador de casos de teste implementados. Para a aplicação da abordagem de validação, primeiramente foi realizada a atividade de **Geração de Casos de Teste** por meio de uma LPS. Em seguida, com os casos de teste produzidos as atividades de **Execução dos Testes** e **Análise de Resultados** foram realizadas nesta ordem. Este ciclo de atividades da VAMT foi realizado novamente, mas na segunda vez foi utilizada a técnica de geração de casos de teste pelo metamodelo.

4.2 Geração dos casos de teste pela LPS

Nesta atividade foi utilizada a LPS de um mini-VANT (Fragal et al., 2013). A Figura 4.1 mostra a arquitetura do sistema especificada por meio de um diagrama de Definição de Blocos com estereótipos da abordagem SyMPLES. Essa arquitetura é composta pelos subsistemas: Comunicação, Sensores, Navegação, Controle de Voo, Atuadores e Carga (*Payload*). Cada subsistema é representado por meio de um bloco com estereótipo *subsystem* do perfil de blocos funcionais da SyMPLES. Os pontos de variação são os blocos com estereótipos do perfil de representação de variabilidades da SyMPLES e indicam que os subsistemas Sensores, Carga e Atuadores possuem variabilidades de configuração.

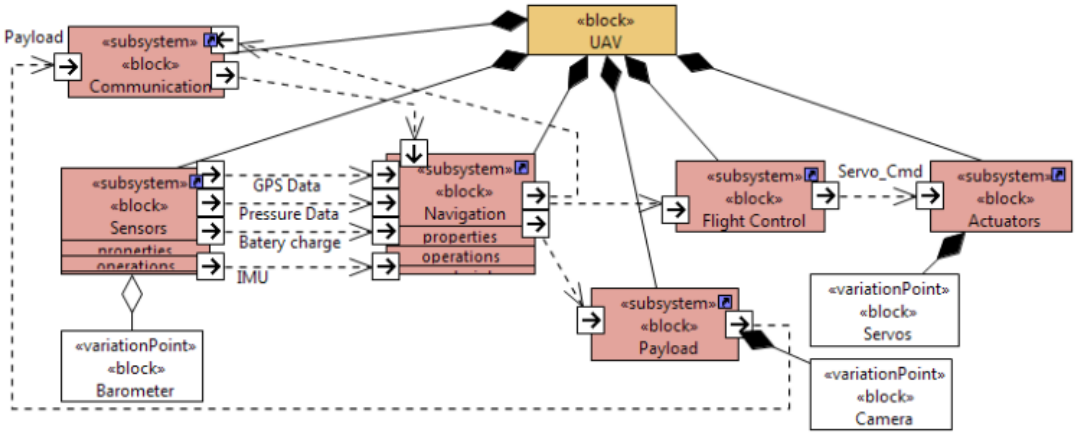

Figura 4.1: Arquitetura do mini-VANT (extraído de Fragal et al. (2013)).

A Figura 4.2 mostra uma análise da LPS utilizada por meio da ferramenta SPLOT. Essa ferramenta requer uma especificação da LPS para realizar as análises. Essa especificação foi criada na própria ferramenta, possui um formato em XML e sua representação visual é mostrada na parte esquerda da Figura 4.2. Com a análise mostrada na Figura 4.2 pode-se identificar a quantidade máxima de modelos configurados que a LPS permite (destacado em vermelho), a quantidade de funcionalidades básicas (que não são variáveis) e a sua consistência.

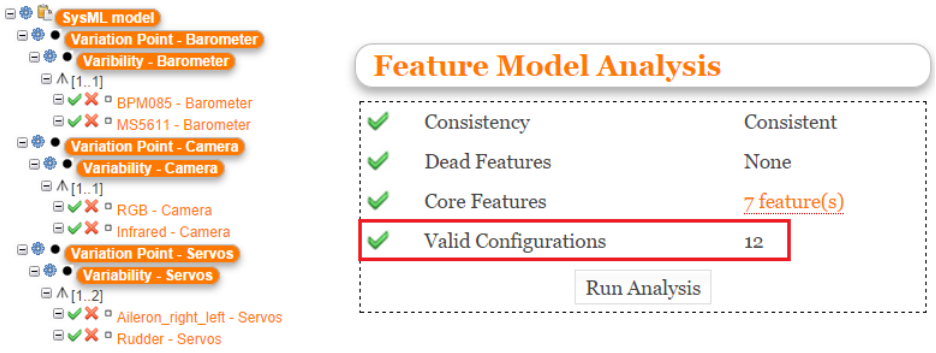

Figura 4.2: Análise da LPS utilizada para o teste da transformação.

O gerador de casos de teste implementado neste trabalho foi utilizado com diferentes critérios de cobertura: parcial, aleatório e total. Todos eles foram usados para efeito de comparação e se baseiam no percentual da quantidade de produtos da LPS. O cálculo da quantidade máxima de produtos que a LPS pode gerar foi realizado com base no BDD apresentado na Seção 2.2, com implementação baseada na ferramenta SPLOT. A Tabela - 4.1 mostra os dados da geração de casos de teste.

Tabela 4.1: Critérios de cobertura utilizados na geração dos casos de teste.

Cobertura	**Critério sobre a LPS (%)**	**Quantidade de produtos gerados**
Total	100	12
Particio-nada	75	9
Particio-nada	50	6
Particio-nada	25	3
Aleatória	8	1

Com o uso do critério de cobertura total, foram gerados 12 produtos no formato VDM da ferramenta pure::variants. Após a escolha das *features*, um *script* é executado para efetivamente configurar cada um dos modelos de acordo com as *features* escolhidas. A partir disso a execução dos testes pode ser iniciada.

4.2.1 Execução dos Testes

A execução dos testes foi dividida em duas etapas, pois a transformação SysML para Simulink é estruturada em duas etapas: a transformação ATL (Etapa 1) e a Geração de blocos funcionais, escrita em linguagem Java (Etapa 2). Mapeando essas etapas na abordagem de validação proposta, as Etapas 1 e 2 correspondem, respectivamente, a **Etapa 2.1** e a **Etapa 2.2**, apresentadas na Seção 3.2.

Quanto maior a quantidade de casos de teste gerados na atividade **Geração de Casos de Teste**, maior o impacto no tempo gasto na execução dos testes. Por esse motivo, a automatização dessa atividade é importante.

No teste da Etapa 1, vários casos de teste mostraram a presença de erros. A Figura 4.3 mostra um comparativo entre a proporção de erros encontrados por casos de teste com a proporção de tipo de erro por casos de teste. Os resultados mostraram que aumentando-se a quantidade de casos de teste houve um aumento na quantidade de erros identificados, mostrado pela curva em azul no gráfico. Os critérios considerados no gráfico são os do tipo particionado e total. No critério aleatório, a escolha do caso de teste foi arbitrária e se baseou no conhecimento do testador sobre o modelo utilizado. Esse critério também mostrou a presença de erro, do mesmo tipo dos demais.

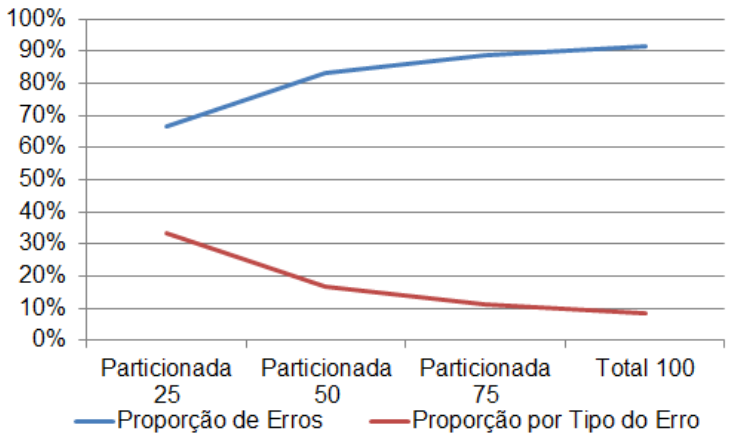

Figura 4.3: Resultados da execução dos testes na Etapa 1 da transformação SysML para Simulink.

Foi observado que com o teste por meio da LPS, neste caso, apenas um tipo de erro distinto foi detectado, e isso resultou na diminuição da proporção de erros encontrados por tipo de erro, mostrado pela curva vermelha no gráfico. Vários erros foram encontrados mas são do mesmo tipo. Nesse caso, a utilização do critério de cobertura "Particionada 25" é o melhor critério, pois obteve a maior proporção por tipo diferente, aproximadamente 33%, excluindo o critério aleatório. Assim, mesmo aumentando-se o número de casos de teste não resultou na identificação de outros tipos de erro. Como o único tipo de erro observado pertence ao tipo 5 da classificação de erros, apresentada na subseção 2.5.1, com a correção dos modelos de entrada e uma nova execução dos testes, nenhum erro foi encontrado. A correção se deu nos modelos de entrada pelo fato da ATL não ter suporte ao tratamento dos modelos de entrada na linguagem.

Por meio dos modelos intermediários XMI gerados, foram iniciados os testes para a Etapa 2, mas também não foram encontrados novos erros.

4.2.2 Verificação do Mapeamento

Terminada a execução dos testes, conforme a abordagem proposta, é necessário verificar o resultado esperado de cada teste. Assim, foi realizada a Verificação de Mapeamento para cada caso de teste executado. Todos os modelos de cada critério de cobertura, totalizando 31 modelos de entrada/saída, foram verificados porém não foi encontrado nenhum erro de mapeamento. Isso indica que os elementos dos modelos de entrada foram corretamente transformados nos modelos de saída para todos os casos.

4.2.3 Teste dinâmico

O teste dinâmico também foi aplicado considerando os modelos SysML gerados pela LPS transformados para *scripts* Matlab/Simulink. Nenhum erro foi encontrado com esse tipo de teste, considerando apenas os 12 modelos de saída do critério de cobertura Total. Alguns *warnings* foram observados, porém não afetaram a execução dos *scripts* que resultam na visualização dos modelos Simulink finais.

4.3 Geração de casos de teste pelo Metamodelo

O gerador de casos de teste baseado em Metamodelo também foi utilizado para a aplicação da abordagem de validação, ressaltando que o metamodelo SysML foi reduzido de acordo com as restrições da transformação SysML para Simulink, conforme apresentado na subseção 3.2.2.

A Tabela - 4.2 mostra uma análise comparativa das duas políticas de geração de casos de teste implementadas. A geração de casos de teste pelo metamodelo apenas considerou os diagramas estruturais SysML: Definição de Blocos, Interno de Bloco e Paramétrico.

Constatou-se que houve uma redução de 75% no número total de casos de teste gerados, utilizando a política N-para-1. Considerando todos os critérios de cobertura, inclusive o aleatório, a redução média foi de 57,86%. Isso possibilitou agilizar o teste, porém um caso de teste gerado na política N-para-1 que identificou um erro pode omitir outros erros. Caso um erro for lançado pela presença de um elemento no diagrama, outros possíveis erros de outros elementos que estão no mesmo diagrama podem ser ocultados.

Um exemplo dessa situação pode ser observado no log de erros apresentado no Apêndice C. Os erros numerados de 1 a 4 na política 1-para-1, obtidos a partir da aplicação dos quatro primeiros casos de teste, não foram encontrados com a política N-para-1. Isso significa que esses quatro primeiros erros foram omitidos, pois o primeiro caso de teste da política N-para-1 mostrou apenas um erro, omitindo os outros quatro erros que foram identificados pela política 1-para-1. Esse exemplo refere-se à aplicação da atividade de teste dinâmico, explicada na Seção 4.3.3.

4.3.1 Execução dos testes

Por meio dos casos de teste gerados, os testes foram executados de mesma forma que a apresentada na seção anterior, divididos em duas partes: teste da Etapa 1 da transformação (ATL) e teste da etapa 2 (Java). A diferença principal é que aqui foram comparadas as duas políticas de geração (N-para-1 e 1-para-1).

Tabela 4.2: Análise da geração de casos de teste para as duas políticas implementadas

Diagrama	**Critério de Cobertura**	**# Casos de Teste**		**Redução nos casos de teste**
		1-para-1	**N-para-1**	
Definição de Blocos	Aleatória (2%)	1	1	0%
	Particionada (25%)	12	3	75%
	Particionada (50%)	25	5	80%
	Particionada (75%)	37	8	78,38%
	Total (100%)	49	10	79,59%
Interno de Bloco	Aleatória (1%)	1	1	0%
	Particionada (25%)	26	6	76,92%
	Particionada (50%)	51	11	78,43%
	Particionada (75%)	77	16	79,22%
	Total (100%)	102	26	74,51%
Paramétrico	Aleatória (3%)	1	1	0%
	Particionada (25%)	8	2	75%
	Particionada (50%)	14	4	71,43%
	Particionada (75%)	25	5	80%
	Total (100%)	33	10	69,70%
TOTAL	—	**184**	**46**	**75%**

A execução dos testes não mostrou erros nas etapas da transformação, e todos os arquivos de saída foram gerados. Assim foi possível iniciar a verificação utilizando os modelos de entrada e de saída para encontrar erros de mapeamento.

4.3.2 Verificação do Mapeamento

A verificação do mapeamento foi realizada considerando os casos de teste gerados. A geração de casos de teste considerou os diagramas estruturais SysML, e a verificação do mapeamento também foi realizada considerando estes diagramas.

A verificação não foi capaz de identificar erros com relação aos diagramas Paramétricos gerados. Isso significa que os 33 modelos paramétricos de entrada foram verificados com os respectivos modelos de saída da transformação e nenhum erro de mapeamento (Tipo 3, apresentado na subseção 2.5.1) foi encontrado. Contudo, na verificação dos diagramas de Definição de Blocos e Interno de Bloco, erros do Tipo 3 foram encontrados. No diagrama

de Definição de Blocos, um caso de teste mostrou a presença de erro desse tipo, e no Interno de Bloco, três. O erro diz respeito aos relacionamentos entre elementos nos modelos de entrada que não foram observados na saída. Dessa forma, a verificação neste caso foi capaz de encontrar erros de Tipo 3, pois o modelo transformado não é produzido com todos os elementos correspondentes aos de entrada.

Um comparativo entre as duas políticas avaliadas, considerando os casos de teste para verificação do diagrama Interno de Bloco, pode ser visualizado na Figura 4.4. Constatou-se que, mesmo reduzindo a quantidade de testes de 102 para 26 (74,5 %), a quantidade de erros encontrada foi igual entre as políticas, mantendo-se em 3 erros encontrados. Assim, a política N-para-1 se mostrou mais eficiente para encontrar erros de mapeamento.

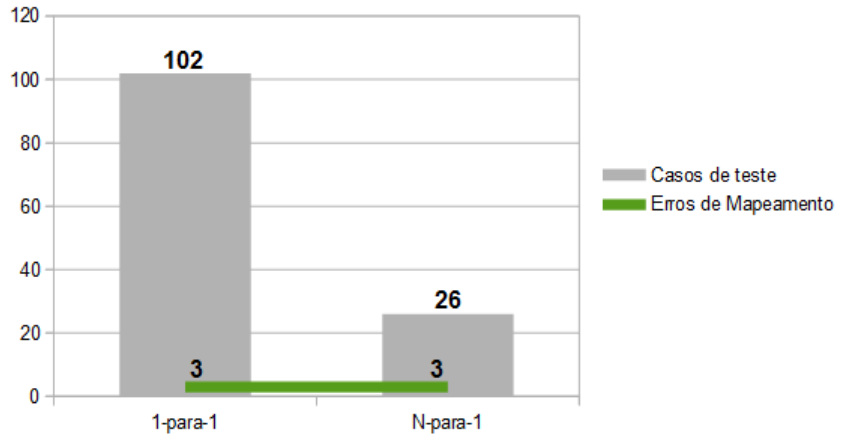

Figura 4.4: Gráfico comparativo entre as políticas 1-para-1 e N-para-1 na verificação de mapeamento dos diagramas do tipo Interno de Bloco.

4.3.3 Teste dinâmico

A aplicação do teste dinâmico considerou as duas políticas de geração de casos de teste. Nenhum erro foi encontrado com esse tipo de teste, para os diagramas de Definição de Blocos e Interno de Bloco. No entanto, surpreendentemente, todos os modelos Paramétricos de saída mostraram erros nesse teste. Isso significa que os *scripts* gerados pela transformação não são capazes de serem executados para a visualização dos modelos Simulink finais. A Figura 4.5 mostra o comparativo entre as políticas de geração avaliadas.

Avaliando-se o segmento de reta em vermelho no gráfico, pode-se constatar que a política 1-para-1 se mostrou mais eficiente nesse caso, pois foi capaz de encontrar 33 erros contra 10 da política N-para-1. Por meio da análise dos arquivos de *log* produzidos no teste dinâmico, foi constatado que o teste dinâmico foi capaz de encontrar erros diferentes,

mas de mesmo tipo (Tipo 3), pois os modelos de saída possuem sintaxe correta mas não foram gerados conforme o esperado pela transformação. Os dois arquivos de *log*, um para cada política de geração para o diagrama Paramétrico, pode ser visualizado no Apêndice C.

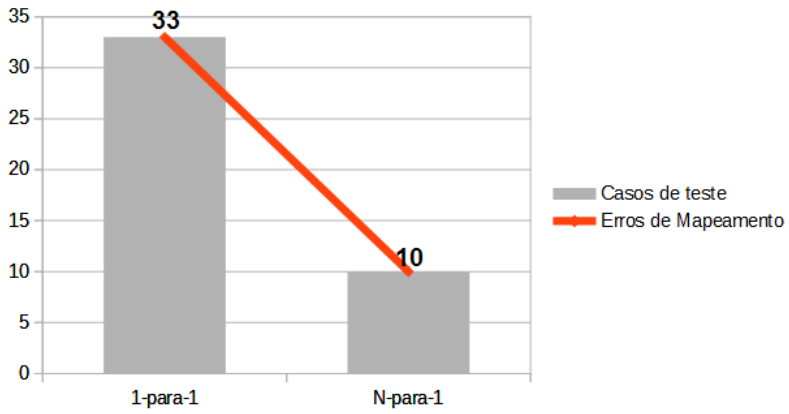

Figura 4.5: Gráfico comparativo entre as políticas 1-para-1 e N-para-1 na verificação de mapeamento dos diagramas do tipo Interno de Bloco.

4.4 Comparação de resultados obtidos

Com a aplicação da abordagem proposta, considerando as duas implementações para a geração de casos de teste, erros foram encontrados na transformação SysML para Simulink.

A Tabela - 4.3 mostra um resumo dos resultados obtidos com a aplicação da VAMT. Os dados se referem aos tipos de gerador implementados, os casos de teste gerados de acordo com a política utilizada, o tipo e a quantidade de erros encontrados, considerando o critério de cobertura Total. Essa tabela mostra o somatório de todos os erros encontrados com a aplicação da abordagem de validação na transformação SysML para Simulink, desde a execução da transformação com os casos de teste gerados, até o teste dinâmico dos modelos de saída.

Apenas dois tipos de erros foram encontrados, o tipo 5 (Codificação incorreta e demais erros) para o gerador pela LPS e o tipo 3 (Modelos semanticamente incorretos) para o gerador pelo Metamodelo. Na Figura 4.6 há um gráfico que mostra a divisão do total de erros encontrados de acordo com os tipos de erro. Pode-se constatar que a maior quantidade de erros encontrados se refere ao tipo 3, obtidos com os casos de teste na geração pelo metamodelo, e assim tomando uma fatia de 82% do total de erros. Nessa perspectiva, a geração pelo metamodelo foi mais eficiente para encontrar erros.

Tabela 4.3: Resumo dos tipos de erros encontrados por tipo de gerador de casos de teste

Gerador	Política de geração	Quantidade máxima de casos de Teste	Tipo de erro encontrado	Quantidade de erros
Gerador pela LPS	-	12	Tipo 5	11
Gerador por meio do Metamodelo	(1-para-1)	184	Tipo 3	37
Gerador por meio do Metamodelo	(N-para-1)	46	Tipo 3	14
TOTAL	-	**242**	-	**62**

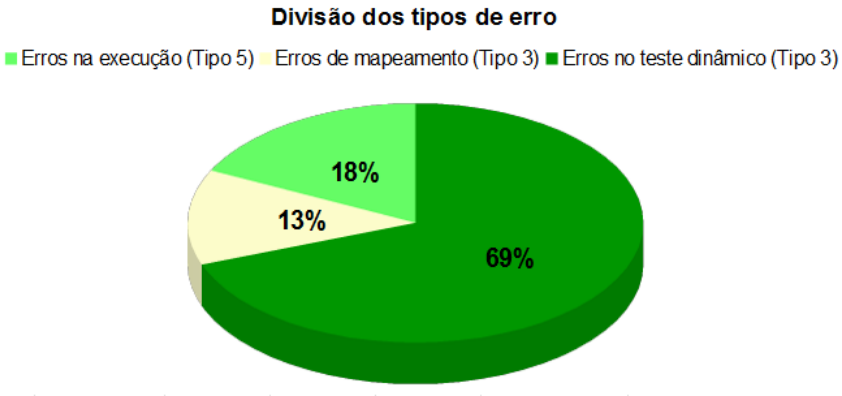

Figura 4.6: Divisão do total de erros de acordo com os tipos de erro encontrados na aplicação da abordagem de validação.

Analisando a proporção da quantidade de erros encontrados por casos de teste gerados, mostrada na Figura 4.7, leva à constatação de que o gerador por meio da LPS foi mais eficiente na validação da transformação. A geração pela LPS obteve 92% na proporção, o que significa que 11 de 12 casos de teste foram úteis para encontrar erros.

Avaliando a VAMT como um todo, a Tabela - 4.3 mostra que foi gerado um total de 242 casos de teste, considerando o critério de cobertura total. Mesmo com a execução, verificação do mapeamento e teste dinâmico, a quantidade total de erros encontrada foi igual a 62, obtendo-se assim uma proporção de erros por casos de teste aproximada em 26%. Esse resultado é satisfatório pois houve redução na quantidade de casos de teste. A quantidade máxima de casos de teste gerados foi reduzida significativamente,

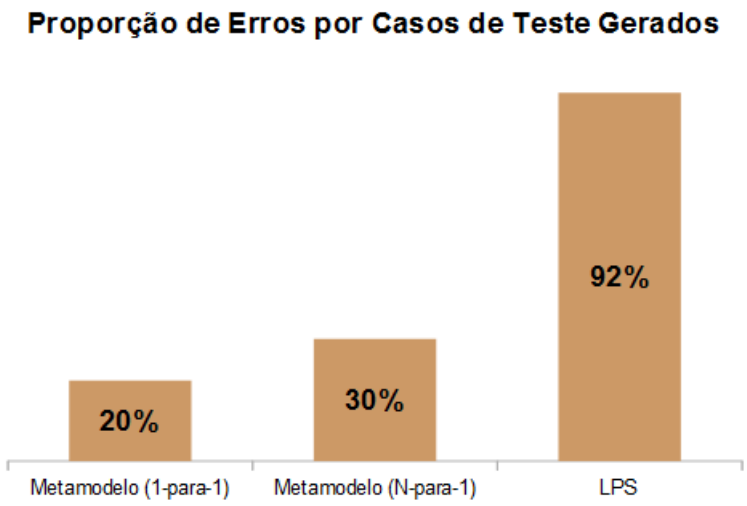

Figura 4.7: Comparativo entre as proporções de erros por casos de teste para os tipos de geradores implementados.

comparando-se com os casos de teste que poderiam ser gerados usando o metamodelo SysML completo. Foi constatada uma redução de 1351 para 184 casos de teste na política 1-para-1 (aproximadamente 86%), permitindo reduzir o tempo total do teste. As subatividades citadas foram automatizadas, ou pelo menos sistematizadas, mas em cada subatividade há dificuldades ou limitações. A seguir são mostradas as ameaças à validade da aplicação da VAMT para a validação da transformação SysML para Simulink.

4.5 Ameaças à Validade da Aplicação da VAMT

Os erros encontrados na geração pela LPS dizem respeito apenas a um problema nos modelos de entrada, que impossibilitou a execução da transformação em sua primeira etapa. Erros de mapeamento utilizando a geração pela LPS não foram encontrados, e também nenhum erro foi encontrado no teste dinâmico. Isso indica que a geração pela LPS pode não ser suficiente para validação da transformação, pois a LPS utilizada é de domínio específico, fato que diminui a quantidade de elementos diferentes testados e além disso ela é pequena em termos da quantidade de configurações possíveis. Outras LPS devem ser testadas para corroborar esses resultados. Por outro lado na geração pelo metamodelo, apenas erros do tipo 3 foram encontrados.

A redução do metamodelo SysML na implementação do gerador também tem impacto na capacidade dos casos de teste gerados em identificar erros de tipos diferentes. Com essa redução eventuais erros do tipo 1 (Cobertura ao metamodelo) não podem ser encontrados.

Assim não há como garantir se todos os elementos do metamodelo SysML foram testados, pois ele foi reduzido de acordo com as restrições da transformação.

Vale ressaltar também que erros do tipo 4 (Ambiguidade) são difíceis de identificar utilizando uma abordagem de teste funcional. Já os erros do tipo 2 (Modelos Sintaticamente Incorretos) requerem conhecimento do metamodelo de saída, embora possam ser detectados também com o teste dinâmico (Tiso et al., 2012).

5

Conclusão

A validação de transformações de modelos se faz necessária para que os modelos sejam transformados conforme esperado. A investigação sobre este tema mostrou que técnicas de teste podem ser utilizadas para a validação de transformações. Nesta dissertação foi proposta uma abordagem de validação baseada em teste funcional para a validação de transformações de modelos, chamada de VAMT. Esta abordagem é constituída por três atividades principais: Geração de Casos de Teste, Execução dos Testes e Análise de Resultados.

VAMT foi aplicada para a validação da transformação de modelos SysML para Simulink da abordagem SyMPLES, sendo que foram usados dois tipos de geração de casos de teste para realizar o teste de transformação, o primeiro é baseado em uma LPS especificada por meio da abordagem SyMPLES e o segundo baseado no Metamodelo SysML. Os dois tipos permitiram a identificação de erros na transformação de forma sistemática, contribuindo para a validação da transformação da SyMPLES. No entanto é necessário que os erros identificados sejam corrigidos na transformação para que a validação seja completa.

Os resultados da aplicação da VAMT na transformação SysML para Simulink indicaram que apenas erros do tipo 5 (apresentado na Subseção 2.5.1) foram encontrados, com relação aos casos de teste gerados por meio da LPS. Com a verificação do mapeamento e teste dinâmico não foi possível determinar a presença de erros na transformação, nesse caso. Assim, há evidências de que usar apenas uma LPS de domínio específico e com poucos casos de teste para a geração pode não ser suficiente para validação da transformação. Pelo fato da transformação SysML para Simulink necessitar os

estereótipos da SyMPLES, dificilmente todos os estereótipos são utilizados e assim nem todos os casos possíveis podem ser testados por meio de uma LPS apenas.

Por outro lado, a geração de casos de teste pelo metamodelo, mesmo reduzindo-o, mostrou que muitos casos de teste foram executados e não encontraram nenhum erro. Utilizando a política N-para-1 para reduzir a quantidade de casos de teste minimizou esse problema, porém ainda assim a proporção de erros encontrados por casos de teste gerados se manteve relativamente baixa. Outras alternativas podem ser exploradas nesse caso, como o uso de heurísticas ou otimizações de algoritmo na geração de casos de teste (Fleurey et al., 2004). Contudo com a aplicação da abordagem foi possível determinar a presença de erros de forma satisfatória, considerando que uma redução significativa na quantidade de casos de teste foi obtida e que todas as atividades foram automatizadas ou sistematizadas para a validação da transformação SysML para Simulink.

5.1 Contribuições deste trabalho

As principais contribuições deste trabalho são:

- Uma abordagem para validação de transformações de modelos, a VAMT. A investigação sobre o tema de validação de transformações mostrou que não há uma abordagem voltada especificamente para transformações divididas em etapas. A VAMT pode ser utilizada independentemente das linguagens utilizadas em cada etapa de transformação pois ela é baseada em teste funcional.

- Um conjunto de ferramentas implementadas para a automatização da VAMT: dois tipos de geradores de casos de teste, um baseado em LPS e o outro no Metamodelo SysML; um programa para verificação de mapeamento; e um *script* para teste dinâmico, todos voltados para a validação da transformação SysML para Simulink.

- Avaliação da VAMT por meio da transformação SysML para Simulink, comparando a quantidade e os tipos de erros encontrados para cada tipo de gerador de casos de teste. Essa avaliação mostrou que foi possível identificar erros na transformação de forma satisfatória e, utilizando políticas de geração e critérios de cobertura, foi possível diminuir a quantidade de casos de teste gerados.

- Adição do diagrama Paramétrico à abordagem SyMPLES, considerando sua importância na especificação de restrições sobre propriedades do sistema.

A abordagem de validação proposta cumpre todas as atividades básicas necessárias para o teste de transformações de modelos. Outras abordagens baseadas em teste de

software propostas em outros trabalhos não consideram todas as atividades de teste de transformações, que iniciam com a Geração de Casos de Teste até a Análise de Resultados. Por exemplo, a abordagem de Lin et al. (2005), ou a de Tiso et al. (2012) assumem que usuário seja responsável pela criação dos modelos de entrada e, portanto, não realizam a geração de casos de teste de forma automatizada. Além disso, as ferramentas elaboradas permitiram a realização do teste estático e do teste dinâmico da transformação, aumentando a quantidade de erros identificados nela. Por meio da identificação dos erros é possível corrigí-los para validar a transformação.

5.2 Limitações

Os resultados obtidos mostraram que a VAMT, embora tenha encontrado erros na transformação, possui algumas limitações. Por ser baseada em teste funcional, o teste de uma etapa depende dos resultados do teste da etapa anterior. Se, em uma etapa, erros impedirem a geração do modelo intermediário, obviamente não é possível testar a etapa seguinte. Portanto, a abordagem é essencialmente sequencial. Uma alternativa é a implementação de um gerador de casos de teste para cada etapa da transformação, podendo, porém, aumentar o tempo da validação.

Outra limitação encontrada é nas implementações realizadas. O gerador de casos de teste pelo metamodelo considerou apenas diagramas estruturais (Definição de Blocos, Interno de Bloco e Paramétrico). O diagrama comportamental (Máquina de Estados) pode ser incluído no gerador. Deve-se considerar também que o esforço para desenvolver ferramentas para geração de casos de teste e para verificação do mapeamento é significativo, e que, na falta de ferramentas para automatização dos testes, pode inviabilizar a aplicação da abordagem.

Pelo fato de ser baseada em teste funcional, erros do tipo 4 (Ambiguidade) são difíceis de encontrar. Normalmente, um mesmo caso de teste não é executado várias vezes para verificar se o resultado obtido com a transformação é o mesmo. Executar o mesmo caso de teste mais de uma vez tem o mesmo impacto que aumentar a quantidade de casos de teste. VAMT poderia ser combinada com teste estrutural, para permitir a identificação desse tipo de erro. Isso poderia melhorar o nível de validação da transformação SysML para Simulink.

A VAMT foi avaliada apenas na validação da trasformação SysML para Simulink, e assim uma avaliação considerando outras transformações se faz necessária. Além disso o gerador baseado em LPS foi utilizado apenas com uma LPS, especificada com a abordagem SyMPLES, que não foi criada com o propósito específico de teste da transformação.

Outras LPS especificadas com estereótipos da SyMPLES poderiam ser utilizadas para o teste da transformação.

5.3 Trabalhos Futuros

Os trabalhos futuros incluem:

- Utilizar outra LPS, também especificada utilizando a abordagem SyMPLES, a fim de corroborar os resultados já obtidos na validação da transformação SysML para Simulink. A LPS utilizada nesta dissertação é pequena em termos da quantidade de possíveis configurações, e possui poucos estereótipos da SyMPLES e portanto uma LPS mais específica para o teste, com mais estereótipos da SyMPLES, deve ser utilizada.

- Avaliar a aplicação da VAMT em outras transformações de modelos, para obter mais informações sobre a sua viabilidade. Na validação da transformação SysML para Simulink, dois tipos de erros foram encontrados, mas não está claro se esses resultados podem ser confirmados em outras transformações.

- Melhorar a abordagem de validação proposta para incluir teste estrutural, pois assim será possível obter um nível maior de validação para as transformações de modelos. Alguns trabalhos, como por exemplo os trabalhos de Küster e Abd-El-Razik (2006) e de González e Cabot (2012), possuem ênfase no teste estrutural.

Outra possibilidade é investigação sobre as técnicas de validação por verificação formal de modelos, usando teste simbólico em máquinas de estados (Guerra, 2012) (Lano et al., 2015) (Gonzalez et al., 2012). Essa técnica apóia a validação e permite avaliar se uma transformação está correta ou não.

Por fim, a revisão bibliográfica realizada neste trabalho de forma assistemática indicou que o estado da arte do tema de pesquisa "Validação de transformação de modelos" necessita de mais pesquisas, por ser um tema relativamente recente, principalmente a validação baseada em teste de *software*.

REFERÊNCIAS

ALMEIDA, P. D. MDA - Model Driven Architecture Improving Software Development Productivity in Large-Scale Enterprise Applications. *University of Fribourg, Switzerland*, 2008.

APACHE Apache ANT Manual. Apache Software Foundation. 2014.
Disponível em `https://ant.apache.org/manual/`

BAUDRY, B.; DINH-TRONG, T.; MOTTU, J.-M.; SIMMONDS, D.; FRANCE, R.; GHOSH, S.; FLEUREY, F.; LE TRAON, Y. Model transformation testing challenges. In: *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, Bilbao, Spain, 2006.

BELATEGI, L.; SAGARDUI, G.; ETXEBERRIA, L. Variability management in embedded product line analysis. In: *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference*, Nice, France: IEEE, 2010, p. 69–74.

BENAVIDES, D.; SEGURA, S.; TRINIDAD, P.; CORTÉS, A. R. Fama: Tooling a framework for the automated analysis of feature models. In: *VaMoS: International Workshop on Variability Modelling of Software Intensive Systems*, Limerick, Ireland, 2007.

BEUCHE, D. Modeling and building software product lines with pure::variants. In: *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, New York, NY, USA: ACM, 2012, p. 255–255 (*SPLC '12*,).
Disponível em `http://doi.acm.org/10.1145/2364412.2364457`

BROTTIER, E.; FLEUREY, F.; STEEL, J.; BAUDRY, B.; LE TRAON, Y. Metamodel-based test generation for model transformations: an algorithm and a tool. In: *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, Raleigh, USA, 2006, p. 85–94.

Büttner, F.; Egea, M.; Cabot, J. On verifying atl transformations using 'off-the-shelf' smt solvers. In: France, R.; Kazmeier, J.; Breu, R.; Atkinson, C., eds. *Model Driven Engineering Languages and Systems*, v. 7590 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 432–448, 2012.
Disponível em `http://dx.doi.org/10.1007/978-3-642-33666-9_28`

Czarnecki, K.; Helsen, S. Classification of model transformation approaches. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Anaheim,USA, 2003, p. 1–17.

Del Fabro, M. D.; Valduriez, P. Semi-automatic model integration using matching transformations and weaving models. In: *Proceedings of the 2007 ACM symposium on Applied computing*, Seoul, Korea, 2007, p. 963–970.

Fleurey, F.; Steel, J.; Baudry, B. Validation in model-driven engineering: testing model transformations. In: *Proceedings of the First International Workshop on Model, Design and Validation.*, Rennes, France, 2004, p. 29–40.

Fragal, V. H. Engenharia de aplicação para sistemas embarcados: transformando especificações SysML em Simulink. 2013. 104f. *Dissertação de Mestrado - Universidade Estadual de Maringá (UEM), Maringá*, p. 104f, 2013.

Fragal, V. H.; Silva, R. F.; de Souza Gimenes, I. M.; de Oliveira Junior, E. A. Application Engineering for Embedded Systems - Transforming SysML Specification to Simulink within a Product-Line based Approach. In: *ICEIS 2013 - Proceedings of the 15th International Conference on Enterprise Information Systems*, Volume 2, Angers, France, 4-7 July, 2013, p. 94–101.
Disponível em `http://dx.doi.org/10.5220/0004402600940101`

Friedenthal, S.; Moore, A.; Steiner, R. *A practical guide to SysML: the systems modeling language.* Elsevier, 2011.

García-Domínguez, A.; Kolovos, D. S.; Rose, L. M.; Paige, R. F.; Medina-Bulo, I. Eunit: A unit testing framework for model management tasks. In: *Model Driven Engineering Languages and Systems*, Springer, p. 395–409, 2011.

Gonzalez, C. A.; Büttner, F.; Clariso, R.; Cabot, J. Emftocsp: A tool for the lightweight verification of emf models. In: *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, Zurich, Switzerland, 2012, p. 44–50.

González, C. A.; Cabot, J. *Atltest: a white-box test generation approach for atl transformations*. Springer, 2012.

González, C. A.; Cabot, J. ATLTest: A White-Box Test Generation Approach for ATL Transformations. In: France, R. B.; Kazmeier, J.; Breu, R.; Atkinson, C., eds. *MoDELS*, Springer, 2012, p. 449–464 (*Lecture Notes in Computer Science*, v.7590). Disponível em `http://dblp.uni-trier.de/db/conf/models/models2012.html#GonzalezC12`

Guerra, E. Specification-driven test generation for model transformations. In: Hu, Z.; Lara, J., eds. *Theory and Practice of Model Transformations*, v. 7307 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 40–55, 2012.

Guerra, E.; de Lara, J.; Kolovos, D. S.; Paige, R. F.; dos Santos, O. M. Engineering model transformations with transml. *Software & Systems Modeling*, v. 12, n. 3, p. 555–577, 2013.

Halim, S. A.; Jawawi, D. N.; Ibrahim, N.; Deris, S. An Approach for Representing Domain Requirements and Domain Architecture in Software Product Line. *SOFTWARE PRODUCT LINE–ADVANCED TOPIC*, p. 23, 2012.

Jouault, F.; Allilaire, F.; Bézivin, J.; Kurtev, I.; Valduriez, P. ATL: a QVT-like transformation language. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, Portland, USA: ACM, 2006, p. 719–720.

Jouault, F.; Kurtev, I. Transforming Models with ATL. In: *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, Berlin, Heidelberg: Springer-Verlag, 2006, p. 128–138 (*MoDELS'05*,).

Kent, S. Model Driven Engineering. In: Butler, M.; Petre, L.; Sere, K., eds. *Integrated Formal Methods*, v. 2335 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 286–298, 2002.

Küster, J. M.; Abd-El-Razik, M. Validation of Model Transformations – First Experiences using a White Box Approach. In: *IN PROCEEDINGS OF MODEVA'06 (MODEL DESIGN AND VALIDATION WORKSHOP ASSOCIATED TO MODELS'06*, Springer, 2006.

Lano, K.; Clark, T.; Kolahdouz-Rahimi, S. A framework for model transformation verification. *Formal Aspects of Computing*, v. 27, n. 1, p. 193–235, 2015.

Lin, Y.; Zhang, J.; Gray, J. A testing framework for model transformations. In: *Model-driven software development*, Springer, p. 219–236, 2005.

Linden, F. J.; Schmid, K.; Rommes, E. *Software product lines in action*. Springer, 2007.

Linhares, M.; Silva, A.; Oliveira, R. Avaliação da SysML através da modelagem de uma unidade experimental de automação industrial. In: *XVI Congresso Brasileiro de Automática - CBL*, Salvador, Brasil, 2006.

Marwedel, P. *Embedded system design*, v. 1. Springer, 2003.

MathWorks Matlab/simulink. 2014.
Disponível em `http://www.mathworks.com/simulink`

Mellor, S. J. *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

Mendonca, M.; Branco, M.; Cowan, D. Splot: software product lines online tools. In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, Orlando, USA: ACM, 2009, p. 761–762.

Novakovic, M.; Passos, L.; Bak, K. Exemplar of automotive architecture with variability report. 2013.

Oliveira Junior, E. A.; Gimenes, I. M. S.; Maldonado, J. C. Systematic Management of Variability in UML-based Software Product Lines. *Journal of Universal Computer Science*, v. 16, n. 17, p. 2374–2393, 2010.

OMG *OMG Unified Modeling Language (UML)*. OMG (*Object-Management Group*), 2011.
Disponível em `http://www.omg.org/spec/UML/`

OMG *OMG Systems Modeling Language (OMG SysML^{TM}) Reference Manual*. OMG (*Object-Management Group*), 2012.
Disponível em `http://www.omg.org/spec/SysML/1.3/`

Oracle Document Object Model (DOM) Java Package Documentation. 2015.
Disponível em `https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html`

Oster, S.; Zorcic, I.; Markert, F.; Lochau, M. Moso-polite: tool support for pairwise and model-based software product line testing. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, Namur, Belgium: ACM, 2011, p. 79–82.

Ostrand, T. J.; Balcer, M. J. The Category-partition Method for Specifying and Generating Fuctional Tests. *Commun. ACM*, v. 31, n. 6, p. 676–686, 1988.

Pohl, K.; Bckle, G.; van der Linden, F. J. *Software product line engineering: Foundations, principles and techniques.* 1st ed. Springer Publishing Company, Incorporated, 2010.

Pressman, R. *Software engineering: a practitioner's approach.* McGraw-Hill higher education. McGraw-Hill Higher Education, 2010.

Reicherdt, R.; Glesner, S. Slicing MATLAB simulink models. In: *34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland: IEEE, 2012, p. 551–561.

Schmidt, D. C. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, v. 39, n. 2, p. 25–31, 2006.

Sen, S.; Baudry, B.; Mottu, J.-M. Automatic model generation strategies for model transformation testing. In: *Theory and Practice of Model Transformations*, Springer, p. 148–164, 2009.

Silva, R.; Fragal, V.; Oliveira-Junior, E.; Gimenes, I.; Oquendo, F. SyMPLES: A SysML-based Approach for Developing Embedded Systems Software Product Lines. In: *Proceedings of the 15th International Conference on Enterprise Information Systems (ICEIS 2013)*, Angers, France, 2013, p. 257–264.
Disponível em `http://hal.archives-ouvertes.fr/hal-00913498`

Silva, R. F. SyMPLES: uma abordagem de desenvolvimento de linha de produto para sistemas embarcados baseada em SysML. 2012. 107f. *Dissertação de Mestrado - Universidade Estadual de Maringá (UEM), Maringá*, 2012.

Tiso, A.; Reggio, G.; Leotta, M. Early experiences on model transformation testing. In: *Proceedings of the First Workshop on the Analysis of Model Transformations*, Innsbruck, Austria: ACM, 2012, p. 15–20.

Warmer, J. B.; Kleppe, A. G. The object constraint language: Precise modeling with uml (addison-wesley object technology series). 1998.

Wu, H.; Monahan, R.; Power, J. F. Metamodel Instance Generation: A systematic literature review. *arXiv preprint arXiv:1211.6322*, 2012.

Apêndice A

Este apêndice mostra o arquivo XMI intermediário produzido pela etapa 1 (Regras ATL) da transformação do diagrama Paramétrico.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:sim="http:///se.kth.md.attest2/Simulink/3.0">
  <sim:Model name="NewDiagram" simulinkName="SysMLmodel/"
  type="BlockDefinition"/>
  <sim:Model name="NewDiagram" simulinkName="Flight Control/SysMLmodel/"
  type="Parametric">
    <parts name="Flight Control" simulinkName="_7twjAHb7EeOea-4BoOCzBw"
    position="x=20 y=10 width=1000 height=600" uuid=
    "_78LZgHb7EeOea-4BoOCzBw" type="Class">
      <children name="compartment_sysml_structure">
        <children name="part:&lt;&lt;moe>> ReadTime" position="x=8 y=171
        width=200 height=100" uuid="_jQ6TQHlkEeO4scp-g95uYQ" type="_kwOJ4HlkEeO4scp-g95uYQ"/>
        <children name="part1:&lt;&lt;moe>>ProcessingTime"
        position="x=733 y=184 width=200 height=100" uuid="_G5LMMHllEeO4scp-g95uYQ"
            type="_JMGREHllEeO4scp-g95uYQ"/>
        <children name="part2:&lt;&lt;moe>>ResponseTime"
        position="x=373 y=35 width=200 height=100" uuid="_MMECIHllEeO4scp-g95uYQ"
            type="_NhkcgHllEeO4scp-g95uYQ"/>
        <children name="constraintproperty:ResponseTime"
        position="x=373 y=265 width=-1 height=-1" uuid="_OhwuMHllEeO4scp-g95uYQ"
            type="_VsIDIHllEeO4scp-g95uYQ">
            <ports name="PrT" position="x=174 y=62 width=20 height=20"
             uuid="_W8KHoHllEeO4scp-g95uYQ" type="constraint_parameter"/>
            <ports name="RespTIME" position="x=80 y=0 width=20 height=20"
             uuid="_XxZtkHllEeO4scp-g95uYQ" type="constraint_parameter"/>
            <ports name="RdT" position="x=-10 y=50 width=20 height=20"
             uuid="_YRrJEHllEeO4scp-g95uYQ" type="constraint_parameter"/>
        </children>
      </children>
    </parts>
    <lines name="ReadTime : Real" type="Connector" source_uml=
    "_YRrJEHllEeO4scp-g95uYQ" destination_uml="_jQ6TQHlkEeO4scp-g95uYQ"/>
    <lines name="ProcessingTime : Real" type="Connector" source_uml=
    "_W8KHoHllEeO4scp-g95uYQ" destination_uml="_G5LMMHllEeO4scp-g95uYQ"/>
    <lines name="ResponseTime : Real" type="Connector" source_uml=
    "_XxZtkHllEeO4scp-g95uYQ" destination_uml="_MMECIHllEeO4scp-g95uYQ"/>
  </sim:Model>
</xmi:XMI>
```

Apêndice B

Este apêndice mostra o *script* Matlab/Simulink que é produzido pela aplicação da etapa 2 (Transformação em Java) da transformação do diagrama Paramétrico.

```
    %Generated M-File by Transformation: SysML Parametric Diagram to Simulink
    %Author: Alexandre Augusto Giron

function SAIDA_MODEL(modelname)
 if nargin == 0
     modelname = 'SAIDA_MODEL';
     modelname_0 = 'SAIDA_MODEL/Flight Control';
     RjQ6TQHlkEeO4scpg95uYQ = 'SAIDA_MODEL/Flight Control/part:<<moe>> ReadTime';
     RG5LMMHllEeO4scpg95uYQ = 'SAIDA_MODEL/Flight Control/part1:<<moe>>Processing Time';
     RMMECIHllEeO4scpg95uYQ = 'SAIDA_MODEL/Flight Control/part2:<<moe>>ResponseTime';

end
open_system(new_system(modelname));
open_system(new_system(modelname_0));
add_block('built-in/SubSystem', RjQ6TQHlkEeO4scpg95uYQ);
set_param(RjQ6TQHlkEeO4scpg95uYQ,'Position', [10 10 65 65 ]);
add_block('built-in/SubSystem', RG5LMMHllEeO4scpg95uYQ);
set_param(RG5LMMHllEeO4scpg95uYQ,'Position', [110 110 165 165 ]);
add_block('built-in/SubSystem', RMMECIHllEeO4scpg95uYQ);
set_param(RMMECIHllEeO4scpg95uYQ,'Position', [210 210 265 265 ]);
%Generating blocks for Constraint Property: constraintproperty:ResponseTime
add_block('built-in/Mux', 'SAIDA_MODEL/Flight Control/Mux_2');
set_param('SAIDA_MODEL/Flight Control/Mux_2', 'Inputs', '2');
set_param('SAIDA_MODEL/Flight Control/Mux_2','Position', [310 310 365 365 ]);
%Port for block RjQ6TQHlkEeO4scpg95uYQ
open_system(RjQ6TQHlkEeO4scpg95uYQ);
add_block('built-in/Outport','SAIDA_MODEL/Flight Control/part:<<moe>> ReadTime/1');
add_block('built-in/Signal Generator','SAIDA_MODEL/Flight Control/part:<<moe>> ReadTime/Signal generator');
%LINE ADD
add_line('SAIDA_MODEL/Flight Control/part:<<moe>> ReadTime','Signal generator/1','1/1');
close_system(RjQ6TQHlkEeO4scpg95uYQ);
%LINE ADD
add_line('SAIDA_MODEL/Flight Control','part:<<moe>> ReadTime/1','Mux_2/1');
%Port for block RG5LMMHllEeO4scpg95uYQ
open_system(RG5LMMHllEeO4scpg95uYQ);
add_block('built-in/Outport','SAIDA_MODEL/Flight Control/part1:<<moe>>Processing Time/1');
add_block('built-in/Signal Generator',
    'SAIDA_MODEL/Flight Control/part1:<<moe>>Processing Time/Signal generator');
%LINE ADD
```

```
add_line('SAIDA_MODEL/Flight Control/part1:<<moe>>Processing Time','Signal generator/1','1/1');
close_system(RG5LMMHllEeO4scpg95uYQ);
%LINE ADD
add_line('SAIDA_MODEL/Flight Control','part1:<<moe>>Processing Time/1','Mux_2/2');

%----------------------------------------
%FCN block section
add_block('built-in/fcn', 'SAIDA_MODEL/Flight Control/ResponseTime = ReadTime + Processing Time');
set_param('SAIDA_MODEL/Flight Control/ResponseTime = ReadTime + Processing Time',
    'Position', [410 410 465 465 ]);
set_param('SAIDA_MODEL/Flight Control/ResponseTime = ReadTime + Processing Time','Expr',' u(1) + u(2)');
%LINE ADD
add_line('SAIDA_MODEL/Flight Control','Mux_2/1','ResponseTime = ReadTime + Processing Time/1');
%----------------------------------------
%Port for block RMMECIHllEeO4scpg95uYQ
open_system(RMMECIHllEeO4scpg95uYQ);
add_block('built-in/Inport','SAIDA_MODEL/Flight Control/part2:<<moe>>ResponseTime/1');
add_block('built-in/Scope','SAIDA_MODEL/Flight Control/part2:<<moe>>ResponseTime/Scope');
%LINE ADD
add_line('SAIDA_MODEL/Flight Control/part2:<<moe>>ResponseTime','1/1','Scope/1');
close_system(RMMECIHllEeO4scpg95uYQ);
%LINE ADD
add_line('SAIDA_MODEL/Flight Control','ResponseTime = ReadTime + Processing Time/1',
    'part2:<<moe>>ResponseTime/1');
```

Apêndice C

Este apêndice mostra dois arquivos de *log* produzidos por meio do teste dinâmico. O primeiro corresponde ao teste dinâmico considerando diagramas Paramétricos produzidos pelo gerador pelo metamodelo, com política 1-para-1, e o segundo se refere à política N-para-1. Esses arquivos mostram a presença de erros nos modelos de saída, e isso indica que a transformação possui erros.

Arquivo 1: *logTesteDinamicoDPp1*

```
------------------------------------- ERRO ENCONTRADO!! -------------------------------------
ERRO 1:
   Modelo de saída:saidaGeracaoParticionada25t39
   Mensagem de erro:Invalid Simulink object name: Mux_0/1
   Linha: 27
------------------------------------- ERRO ENCONTRADO!! -------------------------------------
ERRO 2:
   Modelo de saída:saidaGeracaoParticionada25t40
   Mensagem de erro:Invalid Simulink object name: Mux_0/1
   Linha: 27
------------------------------------- ERRO ENCONTRADO!! -------------------------------------
ERRO 3:
   Modelo de saída:saidaGeracaoParticionada25t41
   Mensagem de erro:A new block named 'saidaGeracaoParticionada25t41/Block/ A = B / C ' cannot be added
   Linha: 23
------------------------------------- ERRO ENCONTRADO!! -------------------------------------
ERRO 4:
   Modelo de saída:saidaGeracaoParticionada25t42
   Mensagem de erro:Invalid Simulink object name: Mux_0/1
   Linha: 20
------------------------------------- ERRO ENCONTRADO!! -------------------------------------
ERRO 5:
   Modelo de saída:saidaGeracaoParticionada25t43
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada25t43
   ')">saidaGeracaoParticionada25t43</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
------------------------------------- ERRO ENCONTRADO!! -------------------------------------
ERRO 6:
   Modelo de saída:saidaGeracaoParticionada25t44
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada25t44
   ')">saidaGeracaoParticionada25t44</a>' because it has been changed.
```

```
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 7:
   Modelo de saída:saidaGeracaoParticionada25t45
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada25t45
   ')">saidaGeracaoParticionada25t45</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 8:
   Modelo de saída:saidaGeracaoParticionada50t84
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t84
   ')">saidaGeracaoParticionada50t84</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 9:
   Modelo de saída:saidaGeracaoParticionada50t85
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t85
   ')">saidaGeracaoParticionada50t85</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 10:
   Modelo de saída:saidaGeracaoParticionada50t86
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t86
   ')">saidaGeracaoParticionada50t86</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 11:
   Modelo de saída:saidaGeracaoParticionada50t87
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t87
   ')">saidaGeracaoParticionada50t87</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 12:
   Modelo de saída:saidaGeracaoParticionada50t88
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t88
   ')">saidaGeracaoParticionada50t88</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 13:
   Modelo de saída:saidaGeracaoParticionada50t89
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t89
   ')">saidaGeracaoParticionada50t89</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 14:
   Modelo de saída:saidaGeracaoParticionada50t90
```

```
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t90
   ')">saidaGeracaoParticionada50t90</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 15:
   Modelo de saída:saidaGeracaoParticionada50t91
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t91
   ')">saidaGeracaoParticionada50t91</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 16:
   Modelo de saída:saidaGeracaoParticionada50t92
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada50t92
   ')">saidaGeracaoParticionada50t92</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 17:
   Modelo de saída:saidaGeracaoParticionada75t131
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t131
   ')">saidaGeracaoParticionada75t131</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 18:
   Modelo de saída:saidaGeracaoParticionada75t132
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t132
   ')">saidaGeracaoParticionada75t132</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 19:
   Modelo de saída:saidaGeracaoParticionada75t133
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t133
   ')">saidaGeracaoParticionada75t133</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 20:
   Modelo de saída:saidaGeracaoParticionada75t134
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t134
   ')">saidaGeracaoParticionada75t134</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 21:
   Modelo de saída:saidaGeracaoParticionada75t135
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t135
   ')">saidaGeracaoParticionada75t135</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
```

```
ERRO 22:
   Modelo de saída:saidaGeracaoParticionada75t136
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t136
   ')">saidaGeracaoParticionada75t136</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 23:
   Modelo de saída:saidaGeracaoParticionada75t137
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t137
   ')">saidaGeracaoParticionada75t137</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 24:
   Modelo de saída:saidaGeracaoParticionada75t138
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionada75t138
   ')">saidaGeracaoParticionada75t138</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 25:
   Modelo de saída:saidaGeracaoParticionado25t38
   Mensagem de erro:Invalid Simulink object name: Mux_0/1
   Linha: 27
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 26:
   Modelo de saída:saidaGeracaoTotal176
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal176
   ')">saidaGeracaoTotal176</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 27:
   Modelo de saída:saidaGeracaoTotal177
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal177
   ')">saidaGeracaoTotal177</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 28:
   Modelo de saída:saidaGeracaoTotal178
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal178
   ')">saidaGeracaoTotal178</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 29:
   Modelo de saída:saidaGeracaoTotal179
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal179
   ')">saidaGeracaoTotal179</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
```

```
ERRO 30:
   Modelo de saída:saidaGeracaoTotal180
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal180
   ')">saidaGeracaoTotal180</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 31:
   Modelo de saída:saidaGeracaoTotal181
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal181
   ')">saidaGeracaoTotal181</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 32:
   Modelo de saída:saidaGeracaoTotal182
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal182
   ')">saidaGeracaoTotal182</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 33:
   Modelo de saída:saidaGeracaoTotal183
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal183
   ')">saidaGeracaoTotal183</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
```

Arquivo 2: *logTesteDinamicoDPp2*

```
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 1:
   Modelo de saída:saidaGeracaoParticionado25t10
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionado25t10
   ')">saidaGeracaoParticionado25t10</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 2:
   Modelo de saída:saidaGeracaoParticionado25t9
   Mensagem de erro:Invalid Simulink object name: Mux_0/1
   Linha: 20
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 3:
   Modelo de saída:saidaGeracaoParticionado50t18
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionado50t18
   ')">saidaGeracaoParticionado50t18</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
```

```
ERRO 4:
   Modelo de saída:saidaGeracaoParticionado50t19
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionado50t19
   ')">saidaGeracaoParticionado50t19</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 5:
   Modelo de saída:saidaGeracaoParticionado75t28
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoParticionado75t28
   ')">saidaGeracaoParticionado75t28</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 6:
   Modelo de saída:saidaGeracaoTotal41
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal41
   ')">saidaGeracaoTotal41</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 7:
   Modelo de saída:saidaGeracaoTotal42
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal42
   ')">saidaGeracaoTotal42</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 8:
   Modelo de saída:saidaGeracaoTotal43
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal43
   ')">saidaGeracaoTotal43</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 9:
   Modelo de saída:saidaGeracaoTotal44
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal44
   ')">saidaGeracaoTotal44</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
-------------------------------------- ERRO ENCONTRADO!! --------------------------------------
ERRO 10:
   Modelo de saída:saidaGeracaoTotal45
   Mensagem de erro:Cannot close the model '<a href="matlab:open_system ('saidaGeracaoTotal45
   ')">saidaGeracaoTotal45</a>' because it has been changed.
   Use the command 'save_system' to first save the model
   Linha: 39
```