Abstract: Model-Driven Engineering (MDE) supports model evolution and refinement by means of model transformations at several abstraction levels. Validating these transformations is essential to ensure the quality and correctness of such models. However, MDE transformations become more complex to validate, for example, when they are implemented in different languages. One particular example is the transformation of the SyMPLES approach. SyMPLES is a development approach for embedded systems, which is based on concepts of both Software Product Lines (SPL) and MDE. SyMPLES has a model transformation process which creates Simulink models from SysML models. This paper presents a case study which applies test case generation based on SPL to validate this model transformation. An SPL was used to generate a set of test cases based on coverage criteria. The results showed that the test cases generated uncovered errors in the transformation of SyMPLES. In addition, a comparison with the test case generation based on metamodel is presented, in order to analyze the effectiveness of the techniques. The coverage criteria made it possible to reduce the number of test cases generated, thus minimizing test effort and time.

Keywords: MDE Validation, Software Product Line, Embedded Systems

Introduction

An embedded system is a computer system incorporated in a larger product (Marvedel, 2003). The development process of these systems is different from traditional systems due to their specific nonfunctional requirements, such as, power consumption, performance, hardware and software integration, real time and cost. Therefore, specific approaches are needed to support their development process.

Model-Driven Engineering (MDE) and Software Product Lines (SPL) are complementary approaches that contribute to improving the embedded systems development process. MDE allows the generation of applications and models by means of model transformations at distinct abstraction levels (Mellor, 2004) and SPL supports non-opportunistic reuse of common artifacts from the same domain to customize new applications (Van der Linden et al., 2007). One of the goals of MDE is to facilitate the code generation from specification models, which can be carried out automatically using MDE transformations.

The SysML-based Product Line approach for Embedded Systems (SyMPLES) (Silva et al., 2013; Fragal et al., 2013) is an approach to support the embedded systems development process, which combines both MDE and SPL concepts. SyMPLES is composed of activities for both variability management and model transformation. It guides the development using annotations in SysML models (Friedenthal et al., 2009) to specify the system. The annotations support the variability resolution to configure specific products.

SyMPLES transforms configured SysML models into Simulink models. SysML is a modeling language for the specification of dynamic systems, at a high abstraction level, whereas MATLAB/Simulink (Mathworks, 2017) is an environment which supports modeling, system simulation and C/C++ code generation.

Validation of MDE transformations is important to ensure quality (Fleurey et al., 2004). If the models are
Another objective in this study is to compare the results of test case generation from the SysML metamodel. Now, model transformation of the SyMPLES was tested using SPL to the validation of the model. The main objective of this paper is to evaluate the test case generation technique using a SPL as input to the test case generation, transformation, the case study involves an MDE validation of the transformed models (output models). It is important to seek alternatives to reduce the size of the test cases, for example using coverage criteria and generation policies, in the context of MDE transformations. Any MDE transformations must be validated to analyze whether its execution produces expected results; and conclusions, contributions and future work are presented in the seventh section.

Related Work

Validation of MDE transformations was investigated in many studies. There is a variety of techniques for validation as Model-Based Testing, formal verification and validation based on common software testing techniques.

Brottier et al. (2006) presented a test case generation process based on metamodel. This process consists on creating partitions of the metamodel, using equivalence classes, aiming to reduce the input domain. These partitions are used to create model fragments that can be used to generate input models for testing. A limitation found is that the scope is reduced to the test case generation. Our work focuses on the test case generation but other activities are also considered (i.e., Run tests activity).

Tiso et al. (2012) provide a development method for MDE transformations and two approaches for testing: Static test and dynamic test. However, no technique for test case generation is discussed or proposed, thus it is assumed the tester already has the input models to the test execution.

Another approach for validation of transformations is the Model-Based Testing (Guerra, 2012; Lano et al., 2015). Differently from common software testing, this kind of testing is usually performed with a symbolic execution of the models. These models are used to specify the desired behavior of the transformation, for example, state machine models.

Lin et al. (2005) proposed a framework and a tool for transformation testing. The tool allows the mapping verification between input/output models and shows the visualization of the differences between these models.

In this study, validation is based on functional testing, mostly because the SysML to Simulink model transformation implementation is composed of two different languages. SyMPLES does not provide any formal model to perform model-based testing of the transformation and this kind of testing is out of the scope of this paper. The main three activities of testing MDE transformations were considered in this
study, from test case generation to the execution and analysis. In addition, a comparison with previous work is performed in this study.

**Test Case Generation based on the SysML Metamodel**

A previous work tackled the validation concern of the SyMPLES model transformation using a metamodel-based test case generation technique (Giron et al., 2017). The previous work technique was applied using the SysML metamodel, with two generation policies and a set of coverage criteria. The generation policies were used to define the size of a test case, because the bigger a test case is, the more difficult to interpret it (Sen et al., 2009). In addition, a bigger set of test cases increase the test effort and time, therefore, an automated environment, an analysis of the SysML metamodel and coverage criteria were used.

The generation policies were applied to define how many elements compose a SysML model. Each SysML model was used to test the model transformation of the SyMPLES approach. With regard to the policies, they are described as follows:

- **(N to 1) policy**: A limited set of SysML elements is inserted in the same diagram. Heuristics can be applied to define the N value. For example, each test case could group at most five of the possibilities from one specific type of relationship between two elements.
- **(1 to 1) policy**: One diagram for each new element. Using this policy it is easier to find out the cause of an error when it occurs, but it could increase the size of the set of test cases.

With regard to the input domain for the test case generation, the coverage criteria determined how much of the SysML metamodel was used. Also, in order to reduce the test case set, a strategy applied was to generate only elements that could be used by the transformation rules. For example, certain diagrams from the SysML language were not taken into account in the SyMPLES model transformation. Therefore, these diagrams and elements were excluded. The result of these strategies was a reduction of the number of test cases and avoided to generate useless test cases.

After running tests using the test cases generated and comparing the policies used, the effectiveness was analyzed. The effectiveness was calculated in terms of how many test cases were able to identify an error in the model transformation. Comparing the policies used, the effectiveness was about 18% (1 to 1) and 22% (N to 1).

A characteristic of the test case generation based on metamodel is the set of test cases are generic. Each test case is composed of a SysML model (and the expected result of the transformation), but this model is generic. This means it does not represent a real system specification, for example. This is not considered to be a problem, however, if the test could be more specific to the model transformation, it would improve the effectiveness of the test case set.

**The SyMPLES Approach**

SyMPLES can be divided into two parts: Domain engineering, which contains the SPL related activities, as variability management; and application engineering, composed of a model transformation.

SPL in SyMPLES is specified with SysML models with stereotypes to variability management and for functional blocks. The transformation of SyMPLES can be applied on the configured SysML models to obtain corresponding Simulink models. SyMPLES uses profiling mechanism for creating two extension profiles of the SysML language, as follows (Silva et al., 2013):

- The SyMPLES Profile for Functional Blocks (SyMPLES-ProfileFB) specifies the types of functional blocks by means of a set of stereotypes. These stereotypes provide additional semantics to the SysML blocks. Therefore, this profile helps to map SysML blocks to Simulink blocks.
- The SyMPLES Profile for Representation of Variability (SyMPLES-ProfileVar): Represents the variabilities of an SPL from a set of stereotypes and aggregate values to the elements of the SysML diagrams. Each product of the SPL is a SysML configured model.

In this approach two processes are defined to the SPL activities, as follows:

- SyMPLES Process for Product Lines (SyMPLES-ProcessPL) consists of activities related to SPL artifacts creation.
- SyMPLES Process for Identification of Variabilities (SyMPLES-ProcessVar) is based in the SMarty approach (Oliveira Jr et al., 2010). It aims to support the SPL variability management, from variability identification to product configuration.

The functional blocks profile was created to support the model transformation in SyMPLES approach. This model transformation maps SysML elements to Simulink blocks. Each SysML model can be transformed to a Simulink model, allowing, for example, the simulation of embedded system specified. Therefore, Simulink models generated from this transformation are closer to...
the system implementation. In addition, Simulink models can be used for code generation, provided by the MATLAB/Simulink tool.

**SysML to Simulink Model Transformation**

The model transformation of the SyMPLES approach is divided into three steps: (i) Configure a SysML model, (ii) run an ATL transformation (Jouault and Kurtev, 2005) and (iii) generate functional blocks, as shown in Fig. 1.

**Configure a SysML Model**

In this step, a SysML model with SyMPLES stereotypes must be configured. The model can be composed of four diagrams: Block Definition, Internal Block, State Machine and Parametric. The root diagram is the Block Definition, used to describe the main blocks of the system. The Internal Block represents the internal relationship of a block based on block instances, therefore one Internal Block diagram can be used for each main block specified in the Block Definition. The behavior can be specified in the State Machine diagram and the Parametric diagram specifies block constraints, values and properties. If a block will be used in the system, or if it is optional, then these variabilities must be resolved to configure the SysML model.

The variabilities are specified in a Block Definition Diagram using variation points, defined in the SyMPLES approach. An example of the definition of an SPL for the Mini-UAV, used in this study, is presented in Fig. 2.

In Fig. 2 three variation points are defined: Barometer, Servos and Camera. SyMPLES defines the variation point stereotype which means that it must be one Internal Block Diagram to specify such variability.

The SysML model with SyMPLES stereotypes can be imported to the pure::variants tool (Beuche, 2012), which creates a Variant Model Descriptor (VDM). In Fig. 3 an example of a configured VDM is shown. The Feature Model in Fig. 3 shows three variabilities: Two options mutually exclusive for the Barometer sensors; two options mutually exclusive for the Camera; and two options for the Servos. The variabilities are resolved in this model, then, they are automatically reflected in an output SysML model.

![Fig. 1: SysML to Simulink model transformation, provided by SyMPLES approach](image-url)
Execute ATL Transformation

This step uses ATL rules to select relevant information about the SysML model, like element attributes, stereotypes and graphic data (i.e., element position and size). Each element with relevant information is stored in one XMI intermediary model. This intermediate transformation using an XMI intermediary model makes this process more flexible to deal with EMF-based editors.

Generate Functional Blocks

This step is implemented in Java and uses the XMI model produced in the previous step and the main file of the SysML model, known as UML file. The UML file must be used because it contains values that are referenced by the XMI model and to retrieve the SyMPLES stereotypes in the SysML model. In this step, a MATLAB script is generated and it represents the Simulink model. With its execution, the Simulink model can be visualized.

The SyMPLES transformation process can be classified as a Model-to-Model (M2M) transformation, receiving as input models the SysML models with SyMPLES stereotypes. The SysML configured model is used as input to the transformation, producing an intermediary file (XMI), which is based on the Simulink metamodel (Biehl et al., 2010). The XMI file is an intermediate model. The final Simulink model is generated only after the “Generate Functional Blocks” step.

SyMPLES Transformation Example

An example of input model can be visualized in Fig. 4. It is composed of a SysML Internal Block Diagram and its Block Definition diagram is the same as in Fig. 2. The Navigation block from the Block Definition Diagram is composed of a block “Yapa2”, which has the subsystem stereotype, defined in SyMPLES. This means that a Simulink subsystem block will be generated after the transformation.

The obtained result after the execution of the transformation is an output model composed of a MATLAB script, known as M-File. Then, this script can be executed in MATLAB to produce the Simulink model. Figure 5 shows the Simulink model.

The SyMPLES transformation can be complex to validate because its implementation is divided into two steps, each one designed in a different language. The first one used ATL language and the functional block generation was written in Java. In addition, the transformation uses the SysML and Simulink metamodels and reads stereotypes information from SyMPLES profiles when it transforms the input models. Therefore, it is important to take into consideration all of these concerns for the validation process of the SyMPLES transformation.
In the previous work, a first attempt to validate the model transformation of SyMPLES was performed. The test case generation technique was based on the SysML metamodel. The test cases generated were generic because the only information used was the SysML metamodel and SyMPLES stereotypes.

In this study, an alternative technique to test case generation is proposed. As the SyMPLES model transformation transforms models from SPL specifications, the main idea is to use an SPL to provide test cases for the model transformation. The details of the proposal evaluated in this case study are explained in the next section.

**Validation of MDE Transformations**

Approaches based on software testing are frequently used in industry and can be applied to validate MDE transformations (Fleurey et al., 2004). Two main types of testing can be applied: Black-box testing, or functional testing, in which the input models are compared to the output models after the transformation; and White-Box testing, or structural testing, in which the internal aspects of the model transformation are analyzed.

As previously mentioned, a transformation can be written in different languages (e.g., ATL, QVT) and even common programming languages. In addition, transformations become more difficult to test when the same transformation is designed into steps, where each one can also be written in different languages, like the transformation of the SyMPLES approach. In these cases, a functional approach is more appropriate.

Regardless the testing type, to validate model transformations at least three steps must be followed (Küster and Abd-El-Razik, 2006): Test Case Generation, in which the test case are generated accordingly to a coverage criterion; Oracle definition, which defines the expected result of a test; and Test Execution, to determine and analyze the testing results.

In the Test Case Generation, two factors must be taken into account: The size of one test case and the size of the set of test cases. The size of one test case is the number of elements of the model. A test case with few elements facilitates both its comprehension and an efficient diagnostic when an error is found. However, decreasing the size of the test case may result in an increase of the set of test cases. When the size of the set of test cases is too big, the test becomes unfeasible (Fawaz et al., 2015). Therefore, reducing the size of the set is important to reduce the test time by using coverage criteria and generation policies (Fleurey et al., 2004; Küster and Abd-El-Razik, 2006).
Considering the Test Execution, Tiso et al. (2012) define two approaches to transformation testing: Checking static properties (static testing) and analyzing the execution (dynamic testing). Static testing refers to verifying properties in the output models, like checking whether certain attributes are present in the output model. On the other hand, dynamic testing refers to analyze the execution of the output model, if it is executable.

The result analysis process from the execution of the tests in model transformations is also important. A transformation is usually based on rules that map elements from input models to the corresponding one in the output models. Thus, a wrong use of the transformation rules may lead to errors, classified as (Küster and Abd-El-Razik, 2006):

- **(Type 1) Metamodel coverage**: The transformation rules have been implemented, but they are not sufficient to map all elements that the metamodel possess. An example is when the rules can only be applied to certain kinds of elements, thus the other kinds of elements are not mapped.
- **(Type 2) Syntactically incorrect models**: When the transformation rule cause generation of an output model that does not comply with the output metamodel.
- **(Type 3) Semantically incorrect models**: When the transformation rules are applied to an input model and the output model is produced syntactically correct, but it does not produce a model with the expected elements. For example, when an input model with elements is transformed but some elements are missing in the output model. Therefore, the output model is not a correct transformation from the input model.
- **(Type 4) Ambiguity**: The same transformation rule produces different results from the same input model.
- **(Type 5) Errors due to incorrect coding**: Included here all the other types of common errors and the codification errors. Examples are the incorrect primitive types (integer, floating point) and memory references out of bounds.

**Test Case Generation Based on SPL Modeled with SyMPLES**

This section presents the test case generation based on SPL. The SPL must be modeled with SyMPLES in order to generate test cases for the model transformation of the SyMPLES.

As the model transformation of the SyMPLES needs a product from a SPL (i.e., a model) to transform it into a Simulink model, a SPL can be used as the “input domain” to generate several products to test if the transformation can produce the expected results (i.e., the Simulink model).

It is worthy to note that, in this case study, the focus is to test the SyMPLES model transformation, not the SPL. This paper will not perform any SPL testing techniques. A premise used in this case study is that the SPL must be modeled with SyMPLES approach and the software components of the SPL are already tested. An analogy that can be made is that the model transformation corresponds to the System Under Test (SUT) and the SPL corresponds to the input data domain of the SUT. Obviously, it is a reduced domain as it discards elements that are not used by the transformation. The SPL provides a family of SysML models as input to test the model transformation.

However, the test case generation using SPL for model transformations has little research available. Possibly, this is due to the fact that this is a particular situation: An MDE transformation with specific characteristics needs validation and the proposal is to use an SPL as a source of test cases, not to perform SPL testing, but to test the MDE transformation in the context of the Embedded Systems and of the SyMPLES approach.

According to Lochau et al. (2012), SPL testing has an issue about the number of products that can be generated from the SPL. This concept is the same when using an SPL as input domain to test the model transformation. If the SPL has too many products then the test could be difficult to manage. To alleviate this problem, a coverage criterion can be applied to the SPL to determine the maximum number of generated models for testing and therefore allowing partitioning the SPL. Other approaches can be used as well, for instance optimization algorithms (Fleurey et al., 2004) or search-based software testing (Anand et al., 2013).

Therefore, to apply the test case generation based on SPL, to test the model transformation of SyMPLES, three definitions are made, as follows:

- SPL must be modeled using SyMPLES. This is a requirement to the transformation itself, otherwise it will not produce any output model.
- Domain definition: The SPL is used to provide the test cases, therefore it is needed to perform a Feature Analysis in the SPL in order to calculate the maximum amount of products available in the SPL. This number reflects the size of the set of test cases.
- Coverage criteria: It is also needed to define the coverage on the test case set, in terms of the quantity of test cases. It is important to highlight that, if increasing the size of the test case set, then it could increase the test effort and time.
The main idea of this technique is to generate specific test cases, related to the SPL used, because this model transformation requires SyMPLES stereotypes. Basically, in SyMPLES the specification of an SPL of a system will have blocks with stereotypes and its variabilities. The stereotypes define which Simulink block will be generated at the output model level. Therefore, using a real world SPL would be possible to generate specific test cases, to test at least the elements used and allowed to be transformed. Then, a test case will be created based on one product of the SPL. The calculation of the amount of test cases for the generation can be performed with the SPL Feature Analysis procedures.

SPL Feature Analysis

The Feature Analysis can be performed based on the Binary Decision Diagram (BDD). Its implementation is based on the SPlot tool, in this study. Comparative studies in feature analysis showed that BDD is efficient in terms of execution time (Mendonca et al., 2009; Benavides et al., 2007).

BDD uses a logic structure to represent a boolean function, composed of decision nodes and terminal nodes (0 and 1). Each node represents a boolean variable and all the paths will lead to the boolean value 1 when the function is true. Connections will lead to left or right and correspond to the value 0 and 1, respectively. Figure 6 presents an example of a BDD and its corresponding Truth Table of a boolean function S. The function can be resolved using the values in the table to traverse the path in the BDD.

The feature model of an SPL can be represented with a BDD. In an example, in an SPL with two mutually exclusive options to configure, the possible configurations can be defined equivalently using a BDD with an XOR function. Options for SPL features in SyMPLES are alternative OR, alternative XOR, optional and mandatory. In summary, SPL constraints are mapped to boolean functions in a BDD.

Implementation Details

The SPL-based test case generation was implemented in this case study using the Java language and an SPL specified using SysML with stereotypes of the SyMPLES approach. The test case generator allows two types of coverage criteria: Partial and total. They are based on the percentage of the maximum amount of products from the SPL, calculated using Feature Analysis previously presented.

Figure 7 shows the process of SPL-based test case generation and the Test Execution. Initially, the SPL definition is taken as input. The SPL definition includes models specified according to SyMPLES and an XML specification. They are used to generate VDM models, allowing the configuration of each product according to the coverage criterion. In addition, the specification in XML format is used because it is compatible with the BDD feature analysis (based on SPlot tool as mentioned before).

A limitation of this implementation is that the VDM configuration must be manual, in the test case generation. Other SPL can be used for test case generation with the tool implemented in this case study, but it must be specified according to SyMPLES, otherwise it would need a tool to automate the generation of the products. The implementation created in this study is specific to the SyMPLES approach, but the concept would be the same for other types of SPLs.

Fig. 6: BDD example
After the VDM generation, feature configuration must be performed, then a script must be executed to create the SysML models, according to the configured VDM. Such script is generated jointly with the VDM. Then, the SysML models generated by the script and corresponding to each VDM can be used as input to the transformation test. Each SysML model represents a configuration of the SPL (i.e., Config. 1 in Fig. 7). This configuration also the expected results of the transformation, but this is not part of the test case generator tool.

In this case study, a table was used to map the input and output elements, using the transformation rules extracted from the SyMPLES documentation. For example, if an input model contains a determined element, in the output model a corresponding element (or elements) must be present. Therefore, it is possible to check if the transformation test has passed or not.

It is worthy to highlight that if an error in the transformation is identified (for example, in the ATL step of the model transformation), then the second step cannot be tested because there is no output model from this step. This scenario is showed in the reports of Fig. 7. If an error is detected in the Run Tests activity, the transformation will not generate the output model. This scenario is not a problem, because the main objective of a test case is to find an error in the transformation.

**Static Test**

The static test is basically composed of the execution of model transformation using test cases generated and analyzing its results. The execution of the tests aims to run the transformation in a suitable environment. Then, one or more intermediary models or output models can be produced, but only if the transformation executes as expected, without any execution errors.

It is important to highlight using proper tools to automate the execution tends to decrease the test time and facilitate the result analysis. The test case generator implemented in this case study also generates an ANT script to run the tests automatically.

**Dynamic Test**

Dynamic test can be applied if the output models produced by the transformation are buildable or executable (Tiso et al., 2012). If the output models do not execute correctly, then the transformation was not able to generate them as expected. Therefore, the dynamic test was included to provide a higher level of validation.

To automate the dynamic test, a MATLAB script was used, developed in previous work. The script executes each output model using the `feval` command. If an error occurs, it is stored in a text file with prefix `logDynamicTest`. The following information composes the log file: Error number (or count), the name of the output model, error message and the line number. This information is needed to help to find the cause of the error in the model. Finding the cause of an error often requires knowledge of its implementation.

**Validation of the SysML to Simulink Model Transformation**

In this section, the case study scenario is presented. The test case generation based on SPL was applied to the model transformation of SyMPLES.

**Hypothesis of this Case Study**

In this case study, the hypothesis is described as follows:

- The test of the model transformation of SyMPLES using an SPL modeled with SyMPLES is more specialized to find errors compared to a metamodel-based technique (of test case generation)
In other words, an SPL-based technique would allow finding more errors in the transformation with fewer test cases generated, when compared with a metamodel-based technique. The activities performed in this case study are described as follows.

**Activities Performed**

Three main activities were performed to validate the transformation of the SyMPLES approach: Test Case Generation, Test Execution (Run Tests) and Result Analysis.

As previously mentioned, for the validation of the SyMPLES transformation the Test Case Generation by SPL was performed. The SPL used as input to the test case generation was the Mini-UAV specification presented in Fig. 2 and 3. An analysis is shown in Fig. 8 related to the variability “Barometer” and “Servos”. The first variability shows two mutually exclusive options: BPM085 and MS5611, mapped to the BDD as a XOR function. The “Camera” variability is also mapped as a XOR function and the “Servos” variability is mapped as an OR function. In Fig. 8, there are two paths that lead to 1-terminal to the XOR structure and three to the OR structure.

The result of the SPL analysis and considering all of the variabilities is a maximum amount of twelve possible configurations: Two for the Barometer, combined with two for the Camera and three for the Servos, as shown in Table 1. Therefore, twelve SysML models can be generated as products to test the model transformation.

Table 2 shows the results from generating test cases using this Mini-UAV SPL. Using one coverage criteria will define how much SPL products (in this case SyML models) will be tested in the transformation. The last column of Table 2 shows the quantity of errors found when the first step (Step 1: ATL transformation) of the model transformation was tested.

Twelve products were generated using the total coverage criterion. Each product is a SysML model composed of one Block Definition, Internal Block and State Machine Diagrams and then used to test the transformation. Then, each model associated with the expected result of the transformation composes the test case.

It is worthy to note that the Run tests activity was divided into two steps. This is due to the fact that the transformation is structured in two steps: ATL transformation (Step 1) and the Generation of Functional blocks, written in the Java language (Step 2). Table 3 presents an example report, related to the results of the test of Step 1, related to the total coverage criteria. The test cases were named variant because each one is a variant configuration of the SPL.

<table>
<thead>
<tr>
<th>Variability</th>
<th>Function</th>
<th>Possible configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barometer</td>
<td>BPM085 XOR MS5611</td>
<td>2</td>
</tr>
<tr>
<td>Camera</td>
<td>RGB XOR Infrared</td>
<td>2</td>
</tr>
<tr>
<td>Servos</td>
<td>Aileron OR Rudder</td>
<td>3</td>
</tr>
<tr>
<td>Total (Combined)</td>
<td>-</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Number of generated products</th>
<th>Percentage of SPL products covered (%)</th>
<th>Errors found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>12</td>
<td>100</td>
<td>11</td>
</tr>
<tr>
<td>Partitioned</td>
<td>9</td>
<td>75</td>
<td>8</td>
</tr>
<tr>
<td>Partitioned</td>
<td>6</td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>Partitioned</td>
<td>3</td>
<td>25</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test case</th>
<th>Result</th>
<th>Error found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant0</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant1</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant2</td>
<td>Pass</td>
<td>-</td>
</tr>
<tr>
<td>Variant3</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant4</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant5</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant6</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant7</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant8</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant9</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant10</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
<tr>
<td>Variant11</td>
<td>Error</td>
<td>org.eclipse.emf.ecore.xmi.UnresolvedReferenceException</td>
</tr>
</tbody>
</table>
Several test cases showed the presence of errors, in the tests of Step 1 of the transformation. Figure 9 presents a comparison between the coverage criteria and provides an analysis of the type of error found. The error rate is calculated based on the amount of errors found divided by the amount of test cases. The error rate by type is calculated based on the quantity of error types divided by the amount of test cases.

Increasing the number of test cases resulted in a gain in the number of errors found (upper curve in the graph, Fig. 9). In other words, the blue curve represents the variation of the number of test cases that showed an error between the coverage criteria. The coverage criteria applied, showed in Fig. 9, was the types classified as partitioned or total. For an example, in the “Partitioned 25” almost 70% of the test cases showed an error.

Although the results showed a high error rate, only one type of error was detected. The type of error identified was the Type 5 of the classification presented in the Fourth section. This is showed by the bottom curve in the graph. Increasing the number of test cases did not result in the detection of other types of errors (Fourth section, Validation of Model Transformations). Therefore, in this case, the “Partitioned 25” criterion is more efficient because its error rate by different type is higher than others, about 33%.

All the errors identified are related to unresolved references of elements in the input models (UnresolvedReferenceException). Therefore, they were classified as type 5: Common coding errors. Only after the correction of the errors, the test was executed again and no error was found in the transformation. Without errors, the intermediary models were successfully generated.

With the intermediary models generated by the transformation, the test of the second step of the transformation (Step 2: Generate Functional Blocks) was performed, but no error was detected, which means that all of the output models were generated.

Then, the validation proceeds with the dynamic test. Each output model was executed with the dynamic test script and no error was identified, allowing the visualization of the Simulink models, ending the Run test activity.

Comparing Results

Summarizing the results from this case study, the proportion of errors found by test case generated obtained was about 92%, considering total coverage of the SPL. This means the effectiveness of the test case set: 11 from 12 test cases were useful to find errors. However, these errors are related to common coding problems. No error was found in the dynamic test. In addition, errors of Type 1-4 (Fourth section), which are specific of the MDE context, could not be found.

Due to the specific nature of this case study, which involves the test of model transformation of the SyMPLES approach and test case generation from an SPL specified with SyMPLES, it is hard to compare results to evaluate the generation technique.
However, despite the differences in the techniques, some concepts are similar when generating test cases. Sen et al. (2009), for an example, presented a test case generation algorithm based on the metamodel. Using a metamodel-based technique, the test case set generated is more generic, normally bigger and they also applied coverage criteria to minimize the test size, but including mutation factors and other strategies. The mutation factors refer to how much distinguish are the test cases in the test case set. A higher mutation factor could increase the probability of the test case find an error (Sen et al., 2009).

On the other hand, using an SPL-based technique the generation is specific to the possible configurations of the products. Therefore, the mutation between the test cases is limited to the quantity of the variabilities in the SPL.

A simple approach to analyze the mutation score between the test cases, in this context, can be done as follows: divide the sum of the variations by the sum of the core elements of the SPL. Considering the SPL specification, composed of a set of diagrams (including the diagram presented in Fig. 2), the sum of core elements in this case study is 7 blocks (in the block definition diagram) plus 21 blocks (in internal diagrams) (Fragal et al., 2013).

From such 28 blocks obtained from the specification, the variabilities are located on only 6 blocks and thus the mutation score in this context would be about 21%. In addition, as the XOR variabilities require one block only in each product, the changes fall, in practice, to 4 blocks with ~14% mutation score. Comparing to the metamodel-based technique (Sen et al., 2009), the mutation score achieved is up to 87%. This can be explained due to the generic and big test case set in this technique.

**Previous Work**

Before presenting the comparison with the previous work, it is worthy to mention that the same version of the SyMPLES transformation was used. Table 4 shows a summary of the results compared with previous work results. The coverage criterion compared is the total coverage, in both techniques. The errors of Type 5 were found applying the test case generation based on the SPL and errors of Type 3 were found using generation based on the SysML metamodel.

In order to compare the effectiveness of the test case sets in both techniques, the graph in Fig. 10 is presented. The graph leads to the conclusion that the generation based on SPL was more efficient in the validation of the transformation. However, only errors of Type 5 were found, which are classified as common coding errors.

On the other hand, the generation based on metamodel was capable to find errors of Type 3, which are more related to the development of the MDE transformations. In this case study, the test case set effectiveness is higher with the SPL-based technique, but the quantity of errors found is higher with the metamodel based technique.

It is worthy to highlight that errors of Type 4 (Ambiguity) are difficult to find using black-box testing approach and errors of Type 2 require information from the output metamodel, although they can be detected using dynamic testing when applicable (Tiso et al., 2012).

**Threats to Validity**

A threat to validity identified is that the SPL used was relatively small (only 12 potential products) in this case study and domain specific, related to the Embedded Systems context. Using total coverage on the SPL was possible, but this coverage could be difficult to apply. For instance, an SPL with n features can yield up to 2n individual systems (or products) (Classen et al., 2011). Examples of bigger SPLs include: Linux 2.6.32.2 kernel, with 6052 features (Peng et al., 2013) and Eclipse SPL, 1024 features (Johansen et al., 2012).

![Fig. 10: Comparison between SPL-based and metamodel-based techniques, regarding the effectiveness of the test case set generated](image)

**Table 4: Comparison of results with the previous work (Giron et al., 2017)**

<table>
<thead>
<tr>
<th>Generation technique</th>
<th>Generation policy</th>
<th>Maximum amount of test cases</th>
<th>Type of error found</th>
<th>Errors found</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPL-based</td>
<td>-</td>
<td>12</td>
<td>Type 5</td>
<td>11</td>
</tr>
<tr>
<td>Metamodel based</td>
<td>1 to 1</td>
<td>184</td>
<td>Type 3</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>N to 1</td>
<td>46</td>
<td>Type 3</td>
<td>10</td>
</tr>
</tbody>
</table>
Conclusions, Contributions and Future Work

The validation of MDE transformations is required for quality assurance. In this study, a case study for validation based on the functional test of a model transformation was presented. An SPL-based technique was used for test case generation and made it possible to identify errors in the transformation, in a systematic way, contributing to the quality of the transformation.

The main contribution of this case study is to provide an evaluation of the technique of the SPL-based test case generation, in the context of MDE transformations. A motivation to the case study was the fact the test cases generated would be more related to the transformation of the SyMPLES approach, which uses SPL concepts. The hypothesis suggested that test would produce better results, compared to other test case generation techniques, like the metamodel-based. However, with specific test cases generated, it was only identified one type of error (Type 5). Although this technique obtained a higher rate of errors identified in the transformation, supporting the hypothesis proposed, more evidence would be necessary to compare and evaluate SPL-based and Metamodel-based techniques.

Another contribution is the validation of the SyMPLES model transformation. The provided information and the tools developed helped to improve the model transformation.

The tools developed in this study aimed to automate the validation activities; however, they are specific to the transformation under test. For example, the test case generation tool can be applied to SPL specified with SyMPLES stereotypes. In general, the concepts are generic but the tools developed are specific to the transformation under test. The high variety of technologies related to MDE transformations can make it difficult to reuse such tools.

Directions for future work would include the investigation of methods of structural testing in the validation of the transformation of the SyMPLES approach. Combining with structural testing the validation level could be higher.

Funding Information

The authors thank CAPES foundation for partially funding this work, to Vanderson Fragal for his support and contributions and to CNPq for the support to the INCT-SEC project.

Author’s Contributions

All authors contribute equally to this work.

Ethics

This paper provides an original contribution of the authors and it is not published elsewhere. All references used are cited in this paper. There is no ethical issue involved in this article.

References


Mathworks, 2017. MATLAB Simulink.