Case Study of Test Case Generation Based on Metamodel for Model Transformations

Alexandre Augusto Giron[1*], Itana Maria de Souza Gimenes[2], Edson OliveiraJr[2]

[1] Federal University of Technology – Parana, Toledo-PR, Brazil.
[2] Universidade Estadual de Maringá, Maringa-PR, Brazil.

* Corresponding author. Email: alexandregiron@utfpr.edu.br
Manuscript submitted February 2, 2017; accepted May 12, 2017.
doi: 10.17706/jsw.12.5.364-378

Abstract: The validation of transformations in the Model-Driven Engineering (MDE) context is important to ensure the quality and correctness of the models. The validation of MDE transformation is burdensome due to both the complexity of the models and the variety of languages that implement them. A test case generation technique can be applied to support the validation process; however, it is challenging because of the size of the test cases set. This paper presents a case study which applies test case generation based on the SysML metamodel for MDE. A development approach for embedded systems, named SyMPLES, was applied. This approach transforms SysML models to Simulink. Two policies were evaluated in the test case generation based on the SysML metamodel. Moreover, a set of strategies and coverage criteria were applied in order to reduce the set of test cases generated and evaluate its effectiveness. The results showed that relevant errors were identified in the model transformation. The use of generation policies also improve the effectiveness of the test case set generated.

Key words: MDE, test case generation, sysml metamodel.

1. Introduction

Embedded systems are applications for processing embedded information in a larger product which is not usually visible to users [1]. The increased computational power of hardware platforms has led to a fast growth rate of embedded software over the last decades; this is mainly due to the transfer of more functionality to software [2]. As a consequence, embedded systems became larger and more complex, thus more demanding in terms of software engineering techniques [3].

The Model Driven Engineering (MDE) and the Software Product Lines (SPL) are complementary approaches that contribute to improving the development of Embedded Systems. MDE supports the generation of applications and models by means of transformation engines at distinct abstraction levels [4], and SPL supports the reuse of common artifacts from the same domain, in order to customize new applications [5]. One of the goals of MDE is to facilitate the code generation from specification models. The code generation can be carried out automatically using MDE transformations.

However, one of the MDE key challenges is to identify effective validation techniques that can ensure the quality of the models. This paper focus on a validation technique for SysML [8] to Simulink [9] model transformation, using test case generation based on the SysML Metamodel, thus contributing to improve the embedded system development process.

The SysML-based Product Line Approach for Embedded Systems (SyMPLES) [6], [7] aims to support the

embedded system development process. It combines both MDE and SPL concepts, composed of variability management and model transformation activities. It guides the development using annotations in SysML models [8] to specify the system. The annotations support the variability resolution to configure specific products.

The configured SysML models can be transformed into Simulink models using SyMPLES. SysML is a modeling language for specification of dynamic systems, at a high abstraction level while MATLAB/Simulink [9] is an environment which supports modeling, system simulation and C/C++ code generation.

The validation of the MDE transformations is important to ensure the quality of the models [10]. If the models are derived automatically by the MDE transformations, then their quality will depend on the correctness of the transformation [11]. SyMPLES is one example of a model transformation that requires validation.

Software testing techniques are used to validate MDE transformations [11], but they are significantly different from testing traditional software. This is due both to the declarative nature of some transformation languages and to the complex structure of the specification models, which can have different types of elements and relationships between them [12]. Furthermore, model transformations can be implemented using different languages as well as be divided into smaller transformation steps, thus increasing its complexity. Therefore, the complexity of the validation process increases because the testing must consider all the transformation steps and their results.

SyMPLES was initially validated based on an application example which was developed to specify a subsystem to Yapa 2 board, responsible for the flight control of a UAV [7]; this was carried out in the context of the Brazilian National Institute of Science and Technology for Critical Embedded Systems (INCT-SEC). However, the need to develop a general purpose validation technique for the SyMPLES model transformation was identified.

The objective of this paper is to present a case study for testing the SysML to Simulink model transformation, using test case generation based on the SysML Metamodel.

This paper is organized as follows: a description of the SyMPLES, its MDE transformation, and concepts on validation are presented in Section 2. Then the test case generation technique is presented in Section 3 and the Test Execution in Section 4. Section 5 presents the application of the technique and its results; the related work is presented in Section 6 and the Conclusions in Section 7.

2. Background

2.1. SyMPLES Approach

SyMPLES can be divided into two contexts: domain engineering, which contains the SPL activities, like variability management; and application engineering, composed of a model transformation.

SyMPLES models are specified in SysML models annotated with stereotypes to represent variability management and functional blocks. SyMPLES transformation is applied on configured SysML models in order to obtain corresponding Simulink models.

SyMPLES uses profiling mechanism for creating two extension profiles to the SysML language, as follows [6]:

- 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.
- 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 addition SyMPLES defines two processes are also defined to the SPL activities, as follows:

- SyMPLES Process for Product Lines (SyMPLES-ProcessPL): this process consists of activities related to SPL artifacts creation.
- SyMPLES Process for Identification of Variabilities (SyMPLES-ProcessVar): based in Smarty approach [13], this process aims to support the SPL variability management, from variability identification to product configuration.

The functional blocks profile was created to support the SyMPLES model transformation. This model transformation maps SysML elements to Simulink blocks. Therefore, Simulink models generated from this transformation are closer to the implementation of the system, thus there is the possibility of code generation and simulation as provided by the MATLAB/Simulink tool.

The model transformation of the SyMPLES can be classified as a Model-to-Model (M2M) transformation and it is divided into three steps: (i) configure a SysML model, (ii) run an ATL [14] transformation and (iii) generate functional blocks, as shown in Figure 1. First, the SysML model is configured and then it is used as input to the ATL transformation. The first output model is an intermediary file (XMI), based on the Simulink metamodel [30]. The XMI file is an intermediate model. The final Simulink model is generated only after the "Generate Functional Blocks" step. Below, the three steps are explained in detail.

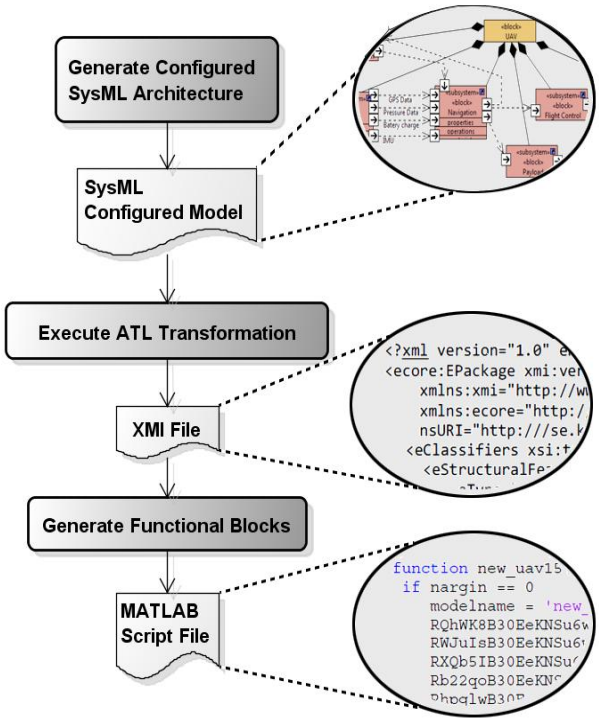

Fig. 1. SysML to simulink model transformation, provided by SyMPLES approach (adapted from [7]).

Configure a SysML model: in this step, a SysML model annotated 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 restrictions, 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 a definition of the SPL used in this paper is presented in Figure 2, called the Mini-UAV [7] [18]. In Fig. 2, three variation points are defined: Barometer, Servos and Camera. SyMPLES approach defines that for each variation point must be one Internal Block Diagram to complete the specification of the variabilities.

The SysML model with SyMPLES stereotypes can be imported to the pure::variants tool [21], which creates a Variant Model Descriptor (VDM). In Fig. 3 an example of a configured VDM is shown. The Feature Model in Figure 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, and then they are reflected in an output SysML model, automatically.

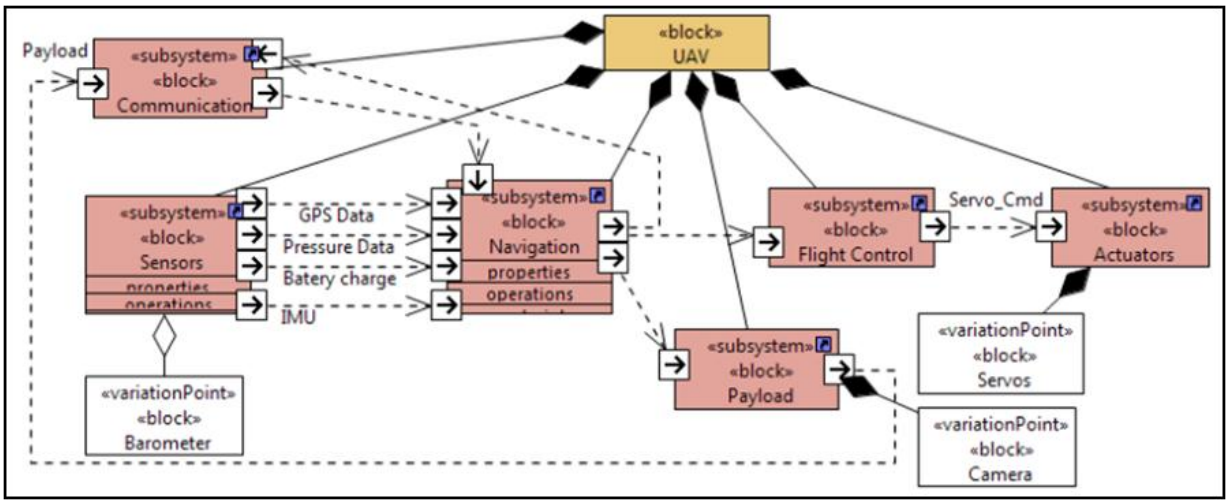

Fig. 2. Block definition diagram of the mini-UAV [7].

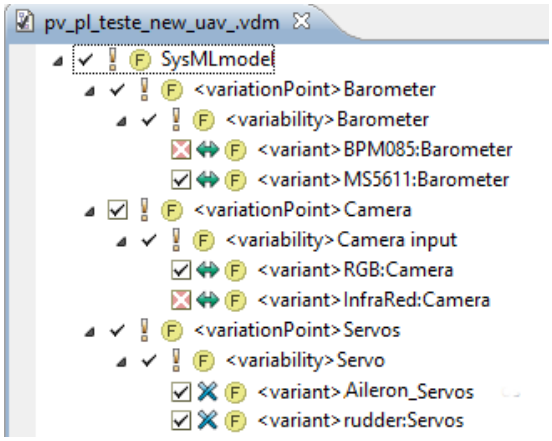

Fig. 3. Feature model of the mini-UAV.

Run 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.

2.2. A SyMPLES Transformation Example

The SyMPLES model transformation requires as input one SysML model annotated with SyMPLES stereotypes otherwise the transformation will be meaningless. An example of a SysML input model can be visualized in Fig. 4. It is composed of an 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 called 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 the MATLAB environment to produce a Simulink model. Figure 5 shows the Simulink model, produced by the transformation.

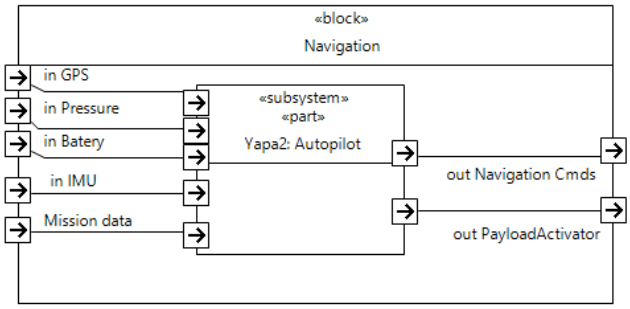

Fig. 4. Example of input model to the transformation of the SyMPLES approach. Adapted from [7].

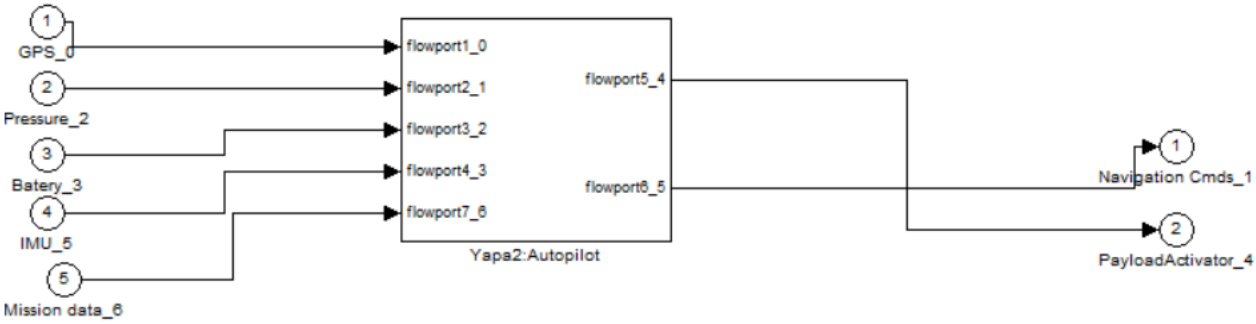

Fig. 5. Example of Simulink model generated by the transformation of the SyMPLES approach. Adapted from [7].

The structure of the transformation shows a complex transformation 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. Validation of model transformations is important because it can improve the quality of the final product [10].

2.3. Validation of Model Transformations

Approaches based on software testing are frequently used in industry and can be applied to validate MDE transformations. 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 internal aspects of the model transformation are considered, for example, the

transformation rules [15].

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

Regardless the testing type, the validation of model transformations requires at least three steps [11]: (i) Test Case Generation, in which the test cases are generated accordingly to a coverage criterion; (ii) Oracle definition, which defines the expected result of a test; and (iii) 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 exhaustive and impracticable [16]. Therefore, reducing the size of the set is important to reduce the test time by using coverage criteria and generation policies [10] [11].

The Test Execution can be carried out using two approaches: checking static properties (static testing) and analyzing the execution (dynamic testing) [12]. Static testing refers to verifying properties in the output models, like checking if some 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 analysis of results from the execution of the tests in model transformations is also important. A transformation normally is 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 [11]:

1) **Metamodel coverage**: transformation rules have been implemented, but they are not sufficient to map all elements of the metamodel. An example is when the rules can only be applied to certain kinds of elements, leaving others unmapped.
2) **Syntactically incorrect models**: when the transformation rule cause generation of an output model that does not comply with the output metamodel.
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.
4) **Ambiguity**: the same transformation rule produces different results from the same input model.
5) **Errors due to incorrect coding**: included here all the other types of common errors and the codification errors. Examples are incorrect primitive types (integer, floating point), memory references out of bounds, among others.

3. Test Case Generation Based on Metamodels

Test case generation based on metamodels generates test cases by extracting all or a set of the possible elements from the metamodel and then creating input models to test the transformation. The generation can produce a very large set of test cases, depending on the metamodel used. To alleviate this problem, coverage criteria and other policies can be applied, reducing the size of the set of test cases.

In this technique, there are three preconditions:

- Define the metamodel to the test case generation: in the transformation of the SyMPLES the SysML metamodel is chosen, but the scope is reduced to the SysML structural diagrams: Block

Definition, Internal Block and Parametric.

- Define the coverage criterion: a coverage criterion must be used to determine how much of the metamodel will be tested. A percentage can be applied to the number of elements and their arrangements in diagrams.
- Choose a generation policy: a type of organization must be defined to the elements and to their relationships in the generated models [11].

Using this technique, the generated models are generic, not from a specific system specification. Therefore, this technique favors the identification of errors of Type 1 to 3, accordingly to the classification presented in section 2.3. Examples include the coverage of the input domain (Type 1) and the element mapping in the transformation (Type 3).

One problem when automating the test case generation is that the generated models may be difficult to interpret by a human tester, depending on the size of the test case used [15]. When a test case shows an error, the tester must understand the model to find out which part, rule or function of the transformation is the cause of that error. In this case, two generation policies will be compared to overcome this problem:

- a set of elements inserted in the same diagram ("N to 1"). Elements with different types are separated and the size of the set is limited. Heuristics can be applied to define the N value. For example, each test case can group at most five of the possibilities from one specific type of relationship between two elements from the metamodel;
- one diagram for each new element ("1 to 1"). Using this policy it is easier to find the cause of an error when it is found.

However, hardly all of the elements from a metamodel are used in a MDE transformation [11]. Thus, a proposal based on metamodel coverage could generate several useless test cases. Heuristics can be applied in the metamodel in order to alleviate this problem.

3.1. SysML Metamodel Analysis for the Test Case Generation

An analysis of the metamodel was carried out to avoid the problem of generating too many test cases that are useless to find errors in the SyMPLES transformation. This analysis allows estimating how much of the SysML metamodel is really used in the transformation. Table 1 shows the number of elements that can be used relative to the total, without considering the relationships between them. For example, the SysML metamodel indicates that 38 different elements can be used in the Block Definition Diagram but only three of these elements are mapped by transformation rules. Thus, the transformation rules map about 60% of the SysML metamodel. In the analysis, the SysML metamodel from Papyrus editor was used (version 0.8.2).

Table 1. Elements from the Metamodel Mapped by the Transformation

Diagram	Elements	Percentage of mapped elements
Block Definition	38	8%
Internal Block	9	56%
Parametric	8	75%
State Machine	13	100%
Average	**-**	**60%**

Another analysis on the SysML metamodel was performed to identify the maximum amount of test cases that could be generated, considering the relationships between the elements of the diagrams, taken two at a time. This analysis is based on the principle that every connection between elements, of any type (composition, dependency, and others), and allowing the repetition, should be tested separately. Thus, the elements were organized in arrangements (A) taken two at a time with repetition. Table 2 shows the

results of that analysis.

Table 2. Analysis of the Maximum Amount of Test Cases, Considering Arrangements from the SysML Metamodel

Diagram	Type of Relationship	Calculation method	# of test cases
Block Definition	Association	$A\ (9,2) = 9^2$	81
	Directed Association	$A\ (9,2) = 9^2$	81
	Composition	$A\ (9,2) = 9^2$	81
	Directed Composition	$A\ (9,2) = 9^2$	81
	Aggregation	$A\ (9,2) = 9^2$	81
	Directed Aggregation	$A\ (9,2) = 9^2$	81
	Dependency	$A\ (16,2) = 16^2$	256
	Generalization	$A\ (10,2) = 10^2$	100
	Interface Realization	Counting	6
	Usage	$A\ (16,2) = 9^2$	256
	Subtotal	-	**1104**
Internal Block	Connector	$A\ (7,2) = 7^2$	49
	Dependency	$A\ (7,2) = 7^2$	49
	Subtotal	-	**98**
State Machine	Transition	$A(10,2) = 9^2 + 20$	120
	Subtotal	-	**120**
Parametric	Connector	Counting	4
	Dependency	$A(5,2) = 5^2$	25
	Subtotal	**-**	**29**
-	**TOTAL**	-	**1351**

In some cases, a counting of the possibilities was performed instead of the arrangement calculation, because not always one element can be connected using a specific relationship. For example, in the Block Definition diagram, in the connection with Interface Realization, only a few elements and in a certain order (source and destination) are allowed.

Adding the arrangements showed in Table 2, the result is the maximum amount of test cases, equal to 1351. This number can be considered high given that the test is not feasible if the execution is not automated.

3.2. Test Case Generation Implementation Details

In addition to the reduced quantity of elements from metamodel used by the transformation, there are specific restrictions embedded in the transformation. These restrictions and design decisions must also be considered to the test case generation. In SyMPLES transformation, five restrictions must be satisfied:

1) The Block Definition and Internal Block diagram requires stereotypes of the SyMPLES profile for functional blocks, which are 48 stereotypes mapping SysML blocks to Simulink blocks. Therefore each one must be tested;
2) Ports must be used to connect blocks;
3) In/out ports are not considered in the transformation;
4) Ports with more than one connection will not be transformed; and
5) The Parametric diagram (if any) must have at least one element of type Constraint Property, to specify mathematics functions of restrictions of the system.

Considering all of these restrictions, the SysML metamodel was reduced but the SyMPLES stereotypes were incorporated into the generation. This means that only models that attend to the restrictions of the transformation will be generated.

Figure 6 shows a pseudocode of the implementation of the generator. The program was written in the Java language and the SysML metamodel from Papyrus editor was used. In addition, the SysML metamodel

was reduced to attend the restrictions of the transformation of the SyMPLES, as previously mentioned.

```
1. Read metamodel
2. Establish generation policy P
3.    if P is 1 to 1 then
4.       for each element E from metamodel
5.          insert E in a new model
6.    if P is N to 1 then
7.       for each set of elements S from metamodel
8.          insert S in a new model
```

Fig. 6. Pseudocode for test case generation.

The two policies cited earlier ("1 to 1" and "N to 1") have been implemented for comparison of the size of the set of test cases generated. Using the "1 to 1" policy means that for each new element E found in the metamodel, a new model will be created to test E in the transformation. Thus, the size of the set of test cases is increased but facilitates the error diagnosis because the test case is simpler.

On the other hand, using the "N to 1" policy the size of the set of test cases tends to be smaller, grouping in each model five different elements, at most. This limit of five elements was fixed to avoid increasing the complexity of the model. After the test case generation, the test execution activity can be performed.

4. Test Execution

The test execution is composed of the Execution and the Dynamic Test. The execution of the tests aims to run the transformation in a suitable environment, using a set of input models previously generated. Then, one or more intermediary models or output models will be produced. It is important to highlight that by using proper tools to automate the execution tends to decrease the test time and to facilitate the result analysis.

```
 8 - diary('logDiaryIBDp1.txt');
 9 - fid = fopen('logDynamicTestIBDp1.txt','wt');
10
11 - countErrors =0;
12 - d = dir([myDir filesep '*.m']);
13 - for jj=1:numel(d)
14 -     [~,name,~] = fileparts(d(jj).name);
15
16 -     try
17          %output model execution
18 -        feval(name);
19 -     catch EXCEPT
20 -             line = EXCEPT.stack.line;
21 -             countErrors = countErrors+1;
22 -             fprintf(fid,'--- ERROR FOUND!! ---');
23 -             fprintf(fid,'\nERROR %d: \n   Output Model:%s\n   Error Message:%s\n   Line: %d\n',countErrors,
24 -     end                                                                                              (...)
25 - end
26 - fclose(fid);
```

Fig. 7. Dynamic test script (excerpt).

The dynamic test can be applied if the output models produced by the transformation are buildable or executable [12]. 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.

The implementation of the dynamic test is written as a MATLAB script. The main part of the script can be visualized in Fig. 7. The script executes each output model using the *feval* command (line 18). If an error occurs, he is stored in a text file with prefix *logDynamicTest*. Any warnings are stored in Diary file. The

following information composes the log file: error number (or count), the name of the output model, the error message and the line number. This information is needed to find the cause of the error in the model. However, in order to identify the error in the transformation, knowledge of the transformation internal structure is required.

5. Results of the Validation of the SyMPLES Model Transformation

The two main activities were performed to validate the SyMPLES transformation, as follows: Test Case Generation (section 3) and Test Execution (section 4).

5.1. Test Case Generation Results

Table 3 shows the results of the test case generation based on the SysML Metamodel, comparing "1 to 1" and "N to 1" policies. In "1 to 1" policy the size of the set of test cases generated is 184, and in "N to 1" the size is equal to 46 test cases. This means a reduction of 75% analyzing the total criterion and, in average, 61.21%. The scope of the implementation of the generator was defined to the SysML structural diagrams.

Analyzing the results for the Block Definition diagram, for example, 49 test cases were generated, considering the total criterion. The transformation of this diagram only considers blocks with SyMPLES stereotypes, which are 48, and only one type of relationship. Additional test cases were not generated because they would not be useful to the transformation under test. The same principle was applied in the generation for the other diagrams.

Table 3. Results of the Test Case Generation, Considering the two Policies Implemented

Diagram	Coverage Criterion	# of test cases		Reduction
		1 to 1	**N to 1**	
Block Definition	Random	1	1	0%
	Partitioned (25%)	12	3	75%
	Partitioned (50%)	25	5	80%
	Partitioned (75%)	37	8	78.38%
	Total (100%)	49	10	79.59%
Internal Block	Random	1	1	0%
	Partitioned (25%)	26	6	76.92%
	Partitioned (50%)	51	11	78.43%
	Partitioned (75%)	77	16	79.22%
	Total (100%)	102	26	74.51%
Parametric	Random	1	1	0%
	Partitioned (25%)	8	2	75%
	Partitioned (50%)	14	4	71.43%
	Partitioned (75%)	25	5	80%
	Total (100%)	33	10	69.70%
-	**Sum (total criterion)**	**184**	**46**	**75%**

5.2. Test Execution Results

After the test case generation, the test execution was performed. The results are divided according to the generation policies and to the coverage criteria. The test execution showed no error in both transformation steps, and all of the output models were created.

Regarding the dynamic test, no errors were found with the Block Definition and Internal Block diagrams. However, all the test cases regarding the Parametric diagram produced errors. An error found in this kind of testing means that the output model could not be executed. Therefore, the Simulink models could not be visualized or loaded for its simulation.

The analysis of *log* files produced during the dynamic test pointed errors of Type 3 (section 2.3). Despite of the fact that the output models are syntactically correct, they were not generated as expected, taking into account the rules of the transformation. For example, a missing *save_system* command was identified in the generated M-File. The "1 to 1" policy lead to 33 errors and 10 errors in the "N to 1", respectively. These results are presented in Fig. 9.

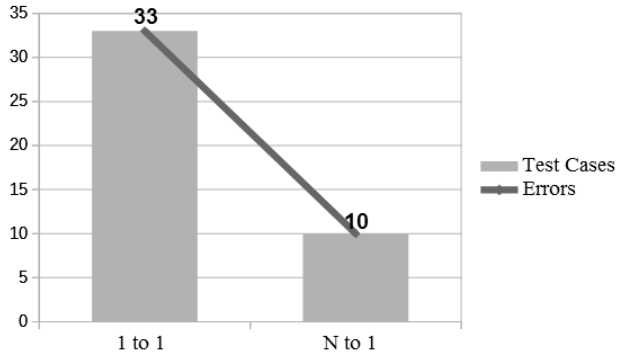

Fig. 9. Comparison between the policies in the dynamic test, regarding the results from parametric diagram test cases.

5.3. Effectiveness

Table 4 presents data that supports the analysis of the effectiveness of the set of test cases generated. It shows a summary of the results considering the two generation policies implemented in our case study as well as the proportion of errors generated in the test cases. This rate suggests a higher effectiveness of the "N to 1" policy. The *log* file of policy "1 to 1" shows that the errors produced are very similar and repeated across the test cases.

Table 4. Summary of the Results Applied to the Transformation of SyMPLES

Generation Policy	Maximum amount of test cases	Type of error found	Amount of errors	Error / Test Cases
1 to 1	184	Type 3	33	17.9 %
N to 1	46	Type 3	10	21.7 %

Only errors of Type 3 were found using this generation technique, independent of the policy applied. The reduction of the SysML metamodel in the metamodel-based generation obviously avoids the production of errors of Type 1 (Metamodel Coverage). Errors of Type 4 (Ambiguity) are unusual because normally the grammar of the transformation languages (or common programming languages) can prevent ambiguity. Therefore, the capabilities of this kind of testing, in this case, are restricted to errors of Type 2, 3 and 5.

6. Related Work

An exploratory study based on interviews and research literature [22] was carried out to investigate methods of quality assurance in QVTo model transformations. One result of this study is a quality model focused on the QVTo transformations, with some properties that could be generalized to other Model-to-Model transformations. Also, a tool to automate the testing was proposed focused on QVTo transformations, but this fact limits the scope to those transformations only.

Brottier et. al. [23] presented a test case generation process based on the metamodel. This process consists of three steps: the first one create partitions of the metamodel using equivalence classes [24], aiming to reduce the input domain; these partitions are used to create model fragments that can be used to

create the models. A limitation found is that the scope is reduced to the test case generation.

Tiso et. al. [12] provides a development method for MDE transformations and two approaches for testing: the static test and the dynamic test. However, no technique for test case generation is discussed or proposed, it is assumed that the tester already has the input models to the test execution.

Other approaches for MDE transformation validation are discussed in [25] and [26]. They are characterized as model-based testing, which uses formal techniques for model checking. Differently from common software testing, this kind of testing is performed based on the symbolic execution of the models generated, normally state machine models. Guerra et. al. [27] presented a family of languages to cover all of the lifecycle of the development of an MDE transformation. A specific language to test case construction is defined but is also based on symbolic execution.

The work proposed by Rocha et. al. [28] presents an approach based on MDE for automating software inspection in structural UML models. Differently from software testing, the main objective of the inspection is to review the software design artifacts, in comparison to a set of models or checklists, in order to find software faults.

Lin et. al. [29] proposed a framework and a tool for transformation testing. The tool allows the mapping verification between input/output models and also shows the differences between these models.

The focus of our case study was the test case generation using the SysML metamodel. In addition, the Test Execution was also considered. These activities aimed to validate a real case scenario, which was the SysML to Simulink model transformation from SyMPLES.

7. Conclusions and Future Work

MDE transformations require validation to ensure that the models are transformed as expected. This paper presented a case study of validation based on the functional test of a model transformation. The technique of test case generation using metamodel was applied to create a set of input models and its execution lead to the identification of errors in the transformation.

The main contributions of this case study are related to the test case generation technique and how it supports validation in the model transformation. Providing test cases that effectively identify errors in the transformation is challenging, in particular when the input domain is big, therefore, resulting in a big sets of test cases.

The technique used the SysML metamodel combined with SyMPLES stereotypes and followed SyMPLES restrictions, so that the test could be more effectively applied. Therefore, the results showed a significant reduction of the set of test cases needed. Comparing the policies related to the size of one test case, the two achieved similar findings (in terms of error rate). The “N to 1” policy shows a higher effectiveness in this case study. However, as “N” defines the size of the test case, a bigger value for “N” turns difficult to find the cause of the error in the transformation, because normally a smaller test case is simpler to understand by a human tester.

The tools developed in this work aimed to automate the validation activities; however, they are specific to the domain of the transformation under test. For example, the test case generator is restricted to the SysML input domain. In general, the concepts presented are generic but the tools developed are specific to the transformation under test. This can be explained due to the variety of technologies related to MDE transformations, turning difficult to reuse those tools. In the other hand, SysML and specially Simulink are very popular in the embedded systems context. Therefore the tools developed in our case study could contribute to the development of embedded systems using SysML and/or Simulink. For example, the Dynamic Test Script can be reused or adapted for other Simulink or MATLAB models.

Some directions for future work would include the evaluation of the structural testing in the validation of

the transformation from SyMPLES approach. Combining the test case generation presented in this paper with structural testing could obtain a higher validation level.

Acknowledgment

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

References

[1] Marvedel, P. (2003). Embedded system design. Springer.

[2] Jerry, R. B., Roberto, P., & Alberto L. S. V. (2001). Using multiple levels of abstractions in embedded software design. *Proceedings of the First International Workshop on Embedded Software (EMSOFT '01)*, Thomas A. Henzinger and Christoph Meyer Kirsch (Eds.). Springer-Verlag, London, UK, UK.

[3] Bassi, L., Secchi, C., Bonfe, M., & Fantuzzi, C. (2011). A sysml-based methodology for manufacturing machinery modeling and design. *IEEE/ASME Transactions on Mechatronics*, *16(6)*, 1049-1062.

[4] Mellor, S. J. (2007). MDA distilled: Principles of model-driven architecture. Addison-Wesley Professional, 2004.

[5] Linden, F. J. V. D., Schmid, K., & Rommes, E. (2007). Software product lines in action.

[6] Silva, R., Fragal, V., OliveiraJr, E., Gimenes, I., & Oquendo, F. (2013). SyMPLES: A sysml-based approach for developing embedded systems software product lines. *Proceedings of the 15 th International Conference on Enterprise Information Systems (ICEIS)*.

[7] Fragal, V. H., Silva, R. F., Gimenes, I. M. D. S., & OliveiraJr, E. A. D. (2013). Application engineering for embedded systems-transforming sysml specification to simulink within a product line based approach. *Proceedings of the 15 th International Conference on Enterprise Information Systems (ICEIS)*.

[8] Friedenthal, S., Moore, A., & Steiner, R. (2011). A practical guide to SysML: The systems modeling language.

[9] *Mathworks.* MATLAB Simulink. 2015.

[10] Fleurey, F., Steel, J., & Baudry, B. (2004). Validation in model-driven engineering: Testing model transformations. *Proceedings of the First International Workshop on Model, Design and Validation, IEEE 2004* (pp. 29–40).

[11] Küster, J. M., & Abd, E. R. (2006). Validation of model transformations – First experiences using a white box approach. *Proceedings of Model Design and Validation Workshop (MODEVA)*.

[12] Tiso, A., Reggio, G., & Leotta, M. (2012). Early experiences on model transformation testing. *Proceedings of the First Workshop on the Analysis of Model Transformations*.

[13] OliveiraJr, E. A., Gimenes, I. M. S., & Maldonado, J. C. (2010). Systematic management of variability in uml-based software product lines. *Journal of Universal Computer Science*, *16(17)*, 2374–2393.

[14] Jouault, F., & Kurtev, I. (2006). Transforming models with ATL. *Proceedings of the International Conference on Satellite Events at the MoDELS (MoDELS)*.

[15] Sen, S., Baudry, B., & Mottu, J. M. (2009). Automatic model generation strategies for model transformation testing. *Proceedings of the Theory and Practice of Model Transformations*.

[16] Fawaz, K., Zaraket, F., Masri, W., & Harkous, H. (2015). Pbcov: A property based coverage criterion. *Software Quality Journal*, *23(1)*, 171–202.

[17] Lochau, M., Oster, S., Goltz, U., & Schurr, A. (2012). Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal*, *20(3-4)*, 567–604.

[18] Fragal, V. H. (2013). Engenharia de aplicação para sistemas embarcados: transformando especificações SysML em Simulink. Master's Thesis, State University of Maringa.

[19] Mendonca, M., Branco, M., & Cowan, D. (2009). SPLOT: Software product lines online tools. *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*.

[20] Benavides, D., Segura, S., Trinidad, P., & Cortes, A. R. (2007). Fama: Tooling a framework for the automated analysis of feature models. *Proceedings of the First International Workshop on Variability Modeling of Software-Intensive Systems*.

[21] Beuche, D. (2012). Modeling and building software product lines with pure::variants. *Proceedings of the 16th International Software Product Line Conference*.

[22] Gerpheide, C., Schiffelers, R., & Serebrenik, A. (2015). Assessing and improving quality of QVTo model transformations. *Software Quality Journal*, *23*, 1–38.

[23] Brottier, E., Fleurey, F., Steel, J., Baudry, B., & Traon, Y. L. (2006). Metamodel-based test generation for model transformations: An algorithm and a tool. *Proceedings of the 17th International Symposium on Software Reliability Engineering*.

[24] Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating fuctional tests.

[25] Guerra, E. (2012). Specification-driven test generation for model transformations. *Theory and Practice of Model Transformations, ser. Lecture Notes in Computer Science*.

[26] Lano, K., Clark, T., & Kolahdouz-Rahimi, S. (2015). A framework for model transformation verification. *Formal Aspects of Computing*, *27(1)*, 193–235.

[27] Guerra, E., Lara, J. D., Kolovos, D. S., Paige, R. F., & Santos, O. M. D. (2013). Engineering model transformations with TransML. *Software & Systems Modeling*, *12(3)*, 555–577.

[28] Rocha, A., Ramalho, F., & Machado, P. (2015). Automating test-based inspection of design models. *Software Quality Journal*, *23(1)*, 3–28.

[29] Lin, Y., Zhang, J., & Gray, J. (2005). A testing framework for model transformations. *Model-driven Software Development*.

[30] Biehl, M., Sjöstedt, C.-J., & Törngren, M. (2010). A modular tool integration approach: experiences from two case studies. *Proceedings of the 3rd Workshop on Model-driven tool and Process Integration*.

Alexandre Augusto Giron is an assistant professor at Universidade Federal Tecnológica do Paraná, Brazil. He completed his master of science in computer science from Universidade Estadual de Maringá, Paraná, Brazil. His research interests include model-driven engineering, software product lines and embedded systems.

Itana Maria de Souza Gimenes is a full professor of software engineering at the Universidade Estadual de Maringá, Paraná, Brazil. She has a post-doctoral research at Open University, UK (2011) where the research was focused on learning design applied to software engineering and a post-doctoral research at the School of Computer Science, University of Waterloo, ON, Canada (2005) where the research was focused on software product line. She has a PhD in Computer Science from the University of York, Department of Computer Science, UK (1992).

Edson OliveiraJr is an assistant professor of software engineering at the State University of Maringá (DIN-UEM), Brazil. He holds a Ph.D. degree in computer science from the University of São Paulo (ICMC-USP), Brazil. He was a visitor scholar at the University of Waterloo (UW), Ontario, Canada in 2009. His research interests include: software product lines, software process lines, variability management, software architecture and evaluation, metrics, model-driven engineering, UML and metamodeling, and empirical software engineering. He acts regularly as a reviewer in journals and conferences, and as a member of the program committee of several international conferences.