Testing of Large Scale Model-Driven Solutions

Bojana Zoranović*, Nenad Todorović*, Željko Vuković*, Aleksandar Lukić*, Gordana Milosavljević*
* Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
{bojana.zoranovic, nenadtod, zeljkov, lukic.aleksandar, grist}@uns.ac.rs

***Abstract*— Testing MDE (Model Driven Development) solutions can be challenging due to their complexity and constant evolution. If the solution is used for product customization of a large scale software product line and introduced in a later phases of development, developing and maintaining testing infrastructure becomes increasingly difficult. In this paper, we examine available techniques for adequate implementation of major test types and present our approach to establishing initial test data, test cases, and validating test results for our MDE solution that supports customization of every layer of a large scale web and desktop business application and code generation of more than 150 different file types.**

Keywords: Model-driven engineering, testing, code generation

I. INTRODUCTION

MDE was developed as a promising approach to address platform complexity and the inability of third-generation languages to alleviate this complexity and express domain concepts by combining domain-specific programming languages with transformation engines and generators [1]. In the context of MDE, we define a system as a "generic concept for designating a software application, software platform or any other software artefact" [2].

The focus of our research are large-scale software product lines (SPLs), and the goal was to use MDE to automatize product creation and customization, along with making maintenance of the existing products easier [3] (Figure 1). Target solution of the SPL was in later phases of development, already delivered to numerous clients and no MDE approach was previously used. Because of the mature state of the SPL target solution, our code generator had to be compliant with existing architecture without introducing changes to the development process. Some of the previous challenges that we encountered during our research, along with results were described in [4].

In this paper, we will focus on examining the techniques and strategies used for testing code generators. Because of the large scale of the SPL, the code generator was being developed in iterations. Even though the domain of testing is mature, such iterative requirement management caused the emergence of testing problems. After each development iteration, new, previously unknown requirements for improvements regarding generated software artefacts were collected from the end-users (developers). Such expansion of the initial solution scope caused changes in the desired contents of the generated files, as well as in the meta-model used to create the specification of the product customization. We found that frequent meta-model evolution brings compatibility issues (every model based on the previous version of metamodel must stay compatible with the new one), along with necessary changes in the generation process. Testing infrastructure must be adaptable to support this constant evolution.

In [5] software testing is defined as "the process of executing a program with the intent of finding errors".

In this paper, we will tackle the following testing techniques:

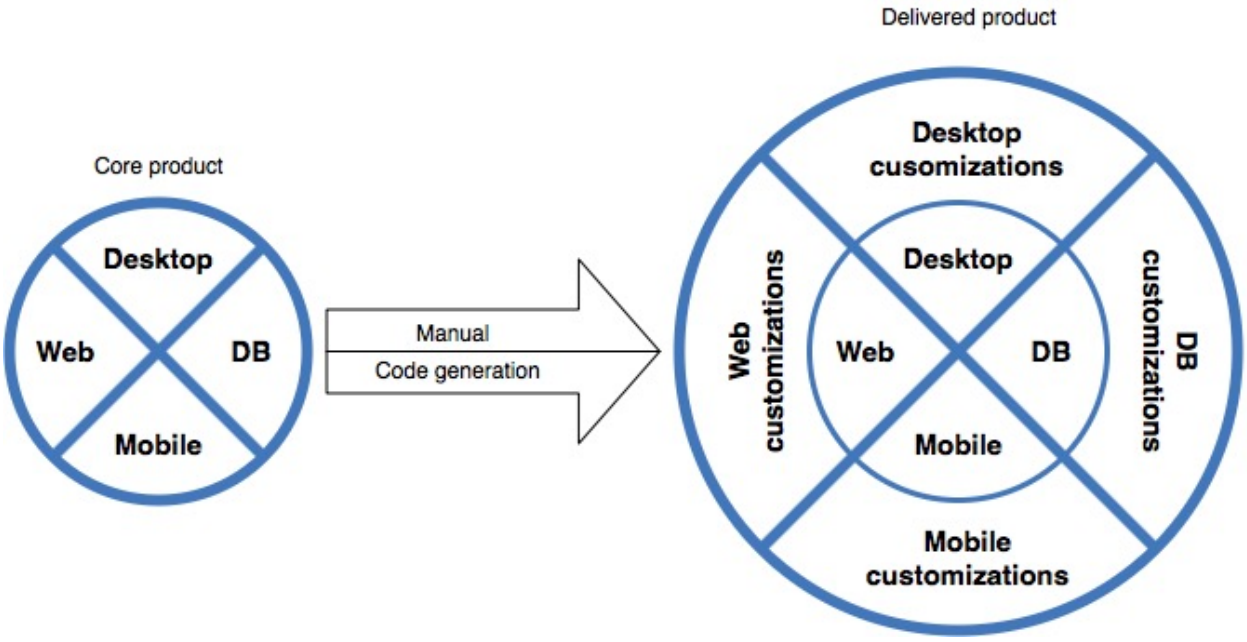

Figure 1. Product customization

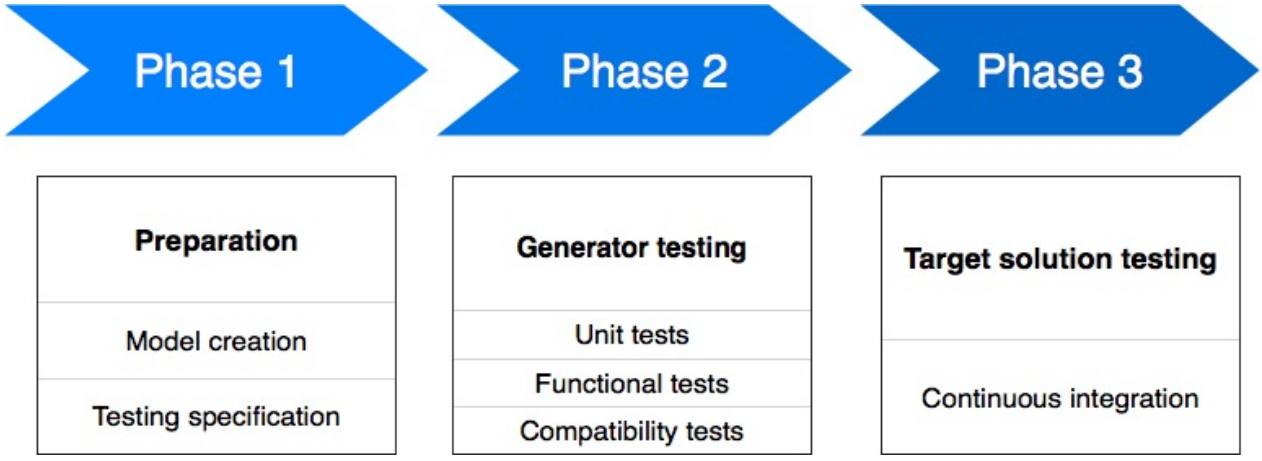

Figure 2. Phases of testing process

- Unit testing (also known as module testing) is a process of testing the individual subprograms, subroutines, or procedures in a program. That is, rather than initially testing the program as a whole, unit testing is focused on the smaller building blocks of the program.
- Function testing is a process of attempting to find discrepancies between the program and the external specification, i.e. precise description of the program's behavior from the point of view of the end user. In our case, the code generator result is generated code - change in the file content.
- Compatibility testing has to assure that new software versions remain backwards compatible.

II. Related work

We sought to find papers discussing different types of test implementation especially in MDE and SPL context.

The first challenge was to generate test data, i.e. test models and define correct (possible) model transformations. Paper [6] suggests that transformations can be defined using general purpose languages, domain-specific languages (such as OCL) or using a rule-based transformation engine. In addition, the language used for transformation should at least support defining pre- and post-conditions.

Paper [7] proposes an algorithm and following tool for automatic test model generation based on the meta-model. The proposed algorithm takes meta-model and a set of model fragments and produces a set of test models. Model fragments can also be derived from metamodel or provided by testers. The algorithm completes each model from the set to become a valid meta-model instance. Although this approach offers a solid basis for generating a broad spectrum of test models, for our setup we considered it time-consuming and complicated since it requires developing a separate algorithm for applying fragments as well as to transform generated test models to valid ones.

The second challenge was to find a suitable method to generate test cases. Some papers advocate validating the behavior of the generation process (model to code transformation) and its properties using formal methods and associated tools [8]. The main drawback of this approach is that it is inefficient for larger models and transformations. An alternative to this is validating transformation only for a set of selected test input models. Although the approach does not prove correctness completely, it effectively identifies emerging problems [9].

The third challenge was to compare generated code with the one that is supposed to be generated. In paper [10], authors based their model comparison on version control systems. This approach uses already existing and stable utilities, however, the comparison does not take into consideration specific language syntax - either primary or secondary (whitespace characters for example) which often leads to detecting false differences.

III. Solution

Our solution constitutes of three phases, depicted in Figure 2:

- In the first phase, the stage is prepared for testing to occur - the test data is scaffolded, and specification for the generator is created;
- In the second phase, the generator is tested with several test types;
- The third test phase serves to test the target solution of the generation process and is part of the already existing continuous integration cycle.

A. Preparation Phase

As has been previously mentioned, the main problem with testing was caused by iterative requirement management, which resulted in frequent changes of the meta-model and expected generated content.

To mitigate the problem of compatibility of test data with the ever-evolving meta-model, we used dynamic model construction - test models are created programmatically instead of being constructed once and loaded from test files when required. We have provided an API to help rapidly scaffold initial test model which ensures that the scaffolded model is a valid instance of the meta-model. After that, test cases simulate user's interaction with the model and iterative nature of code generator usage through the API,

i.e. frequent model to code transformations separated by only a few model changes.

In order to track what content should be generated, we have created a test scenario configuration, which includes a list of files that are expected to change. For each file, we can configure if whitespace character differences should be ignored. Some sensible defaults are provided for this option, based on the type of the file (e.g. leading tabs or spaces are part of the style in Java or C#, but are a part of the syntax in Python). However, in our experience, there are cases where maintaining file layout is required, even if the whitespace characters are not part of the concrete syntax, in order to increase readability or to generate code that resembles handwritten. For this reason, ignoring of whitespaces can be configured per each file.

B. Generator Testing Phase

After the preparation phase, the following test types were developed to detect possible errors in the generation process and generated content.

Unit testing. In order to be tested, the code generator should be modularized into manageable units so that every module is validated separately [11]. When it comes down to unit testing, an examined module should be evaluated detached from the rest of the system. This process requires mock objects to imitate existing application infrastructure that tested module interacts with, along with corresponding interfaces.

Functional testing. Two main example scenarios for functional testing are:

- The code is generated utilizing the model previously created using API. After that, the subsequent generation process is executed without any changes applied to the model. We expect that no modification is introduced with the second generation cycle. This scenario is critical to ensure that the generation process is idempotent;
- The code is generated once, the subsequent changes are applied to the model, and then the code is generated again. In this scenario, we expect that only files affected by the change to the model are modified.

Since the generation process affects many files, it was proven challenging to effectively, reliably and rapidly perform detailed comparisons. Instead, the comparison was performed in several steps:

1. Creation of affected files snapshot. Content and computed hash value are stored for every file (Figure 3);

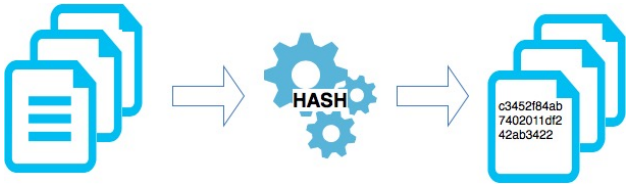

Figure 4. Hashing file content

2. Two generation cycles snapshots are compared to conclude whether there were files created (Figure 4) or deleted (Figure 5) between two generation cycle;

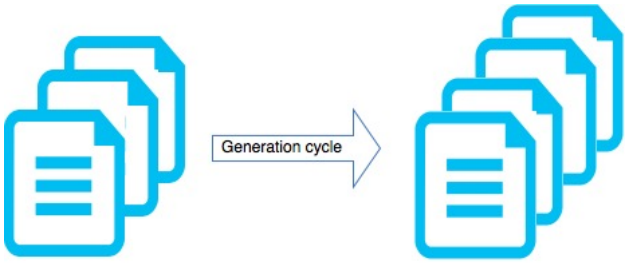

Figure 3. New files added after generation cycle

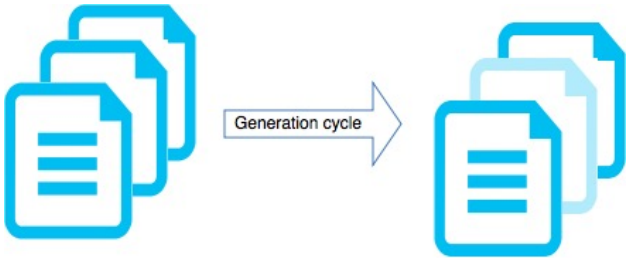

Figure 5. File deleted after generation cycle

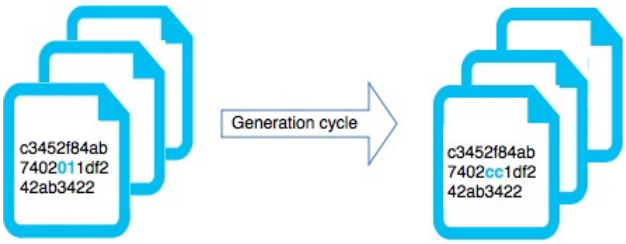

Figure 6. Content hash comparison

4. If there are differences between hash values, content is compared (Figure 7).

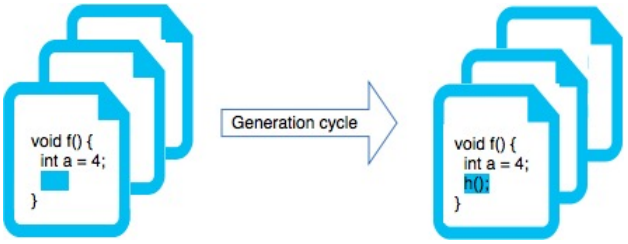

Figure 7. File content comparison

This process consumes the least effort since every action is taken only if it is indispensable and the comparison of the entire content is avoided.

Compatibility testing. Compatibility tests were necessary in order to assure that two different tool versions generate similar output, at least to some extent, considering new features could be introduced in new versions. Code is generated from the same model using two different tool versions.

In addition to the list of files expected (not) to change, the configuration for this test includes references to two version control commits. When the test is executed, the first commit is checked out, the tool is compiled, and code generation is executed. The same is repeated using the latter commit and snapshots of generated content are compared.

The main challenge here is that both tool versions should be run on the same specification, even if the way that the tool stores that specification might have changed between the two versions. For this reason, the specification used for tests is constructed at test runtime. The tool provides an API for building such a specification in the manner specific to its version.

C. *Target Solution Testing*

Finally, in the third and last phase, the code generated from the test model is combined with the rest of the product line code, and the result is submitted to the continuous integration (CI) infrastructure. There, the project is built, and another batch of tests are executed, that now target the end application. These tests are the same ones that are used when application development is performed manually. As this procedure can take a long time, it is usually executed as a part of the nightly CI cycle.

IV. DISCUSSION AND FURTHER WORK

Testing code generators can often significantly increase cost and effort comparing to testing non model-driven solutions [10]. Additionally, test case production is often ad-hoc, manually written or difficult to evaluate [12]. Even with all the challenges, testing MDE solutions is crucial for the reliable development process. MDE solution often starts with automatization of some minor part of problem domain by developing proof of concept. If the proof of concept is accepted, solution scope gradually expands. Creation of every new generator feature must not affect its existing features (in our case, code generator affected more than 150 different software artifacts) which is practically impossible without a wide variety of different test types.

In the near future, we plan to improve a few of the described processes. Firstly, the test model generation could be improved by introducing a rule engine or domain-specific language for a more efficient test model specification. In addition, metamodel evolution imposes occasional changes onto the existing test suite. We aim to automate this process in order to avoid manual test adjustments.

REFERENCES

[1] Schmidt, D.C. Guest editor's introduction: Model-driven engineering. Volume: 39, Issue:2, pp. 25-31, Computer, IEEE 2006

[2] da Silva, A.R., Model-driven engineering: A survey supported by the unified conceptual model, Volume: 43, Issue: C, pp. 139-155, Computer Languages, Systems and Structures, Elsevier Scence publishers, 2015

[3] Völter M., Groher, I., Product Line Implementation using Aspect-Oriented and Model-Driven Software Development, 11th International Software Product Line Conference, IEEE, 2007

[4] Todorović, N., Lukić, A., Zoranović, B., Vaderna, R., Vuković, Ž., Stoja, S. RoseLib: A Library for Simplifying .NET Compiler Platform Usage, ICIST 2018 Proceedings Vol.1, pp.216-221, 2018

[5] Myers, G.J., Sandler, C., The Art of Software Testing, Second Edition, John Wiley & Sons, Inc, USA, 2004

[6] Fleurey, F., Steel, J., Baudry, B., Validation in Model-Driven Engineering: Testing Model Transformations, Proceedings. 2004 First International Workshop on Model, Design and Validation, IEEE, 2004

[7] Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y., Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool, 17th International Symposium on Software Reliability Engineering, IEEE, 2006

[8] Loli Burgueño, L., Wimmer, M., Troya, J., Vallecillo, A.,TractsTool: Testing Model Transformations based on Contracts, CEUR Workshop Proceedings, vol. 1115, pp.76-80, 2013

[9] Cabot, J., Clarisó, R., Guerra, E., de Lara, J., Verification and validation of declarative model-to-model transformations through invariants. JSS 83(2), pp.283–302, 2010

[10] Lin, Y., Zhang, J., Gray, J., Model comparison: A key challenge for transformation testing and version control in model driven software development, Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development, pp.219-236, Springer, 2004

[11] Stürmer, I., Conrad, M., Dörr, H., Pepper, P., Systematic Testing of Model-Based Code Generators, IEEE Transactions on Software Engineering, Volume: 33, Issue: 9, IEEE, 2007

[12] Fleurey, F., Breton, E., Baudry, B., Nicolas, A., Jézéquel, J.-M., Model-Driven Engineering for Software Migration in a Large Industrial Context. Lecture Notes in Computer Science, pp.482–497, MODELS 2007: Model Driven Engineering Languages and Systems, Springer, 2007