Comparing model coverage and code coverage in Model Driven Testing: an exploratory study

Domenico Amalfitano*, Vincenzo De Simone*, Anna Rita Fasolino*, Vincenzo Riccio†
*Department of Electrical Engineering and Information Technologies
University of Naples Federico II
Via Claudio 21, Naples, Italy
Email: {domenico.amalfitano, vincenzo.desimone2, fasolino}@unina.it
†CeRICT - Centro Regionale Information and Communication Technology
Complesso Universitario di Monte Sant'Angelo - Fabbricato 8b,
Via Cintia, Naples, Italy
Email: vin.riccio@gmail.com

***Abstract*—The Model Driven Architecture (MDA) approach is emerged in the last years as a novel software design methodology for the development of software systems. In this approach the focus of software development is shifted from writing code to modeling. In MDA, developers implement models that are automatically transformed into the target code of the system. Alongside MDA, the Model Driven Testing (MDT) is emerging as a relevant research topic in both industrial and scientific communities. MDT is a methodology where test cases for the system are automatically obtained starting from test models to maximize specific model coverage criteria. Eventually, test cases are executed to verify the system code that is generated through an MDA approach. In this paper, we conduct an exploratory study in order to evaluate the differences that may exist between the model coverage guaranteed by the test cases and the code coverage reached when they are executed on the auto-generated code. Moreover, we identify the main factors that may influence these differences.**

I. Introduction

Nowadays software industries are required to develop more and more complex software systems, while respecting shorter and shorter delivery schedules. At the same time these systems are required to satisfy strict quality attributes of safety and security. These trends motivated the request for finding more effective and alternative solutions for software system design, development and testing.

In the last years Model-Driven Engineering (MDE) has emerged as a promising approach for developing software systems that gives particular emphasis on models and automated code generation. The basic difference between MDE and more traditional software development approach is that MDE exploits models to automatically generate the specified software system. MDE is based on (1) *domain-specific modeling languages* and (2) *transformation engines and generators* [1]. Domain-specific modeling languages are exploited for formalizing the software structure, behavior, and requirements within particular domains, such as software-defined radios, avionics mission computing, automotive safety software, online financial services, warehouse management, or even the domain of middleware platforms. Transformation engines and generators analyze certain aspects of the models and then generate various types of artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations. Currently a dominant approach in MDE is the Model Driven Architecture (MDA)[1], an initiative of the Object Management Group (OMG) that exploits OMG UML models in the development process.

The same emphasis on modeling that underlies MDA can be found in Model Driven Testing (MDT) too [2]. In MDT, test cases that exercise the code of the system under test are automatically obtained starting from test models. They are usually generated by techniques aimed at reaching an adequate coverage of the test model elements. However, even if test cases adequately cover the test models, it is not true that they will be sufficient to reach an adequate coverage of the system code, too. Investigating the relationship between model coverage and code coverage of test suites obtained by a MDT approach is hence a relevant research topic. Although this problem was addressed in other model driven approaches like the Model Based Development (MBD) one [3], it has not been sufficiently investigated in the MDA context. In this paper, we are interested to explore this relationship in the context of MDA approaches. In particular, we intend to study an MDT approach that exploits UML StateMachines both to specify the software system and to define the test model.

The rest of the paper is organized as follows in Section II a background about model driven approaches is reported. Section III describes the exploratory study we performed and Section IV summarizes the findings of the study. Eventually, Section V reports conclusions and future works.

II. Model Driven Testing Background

The OMG's **Model Driven Architecture**, MDA, initiative shifts the focus of software development from writing code to modeling [4].

MDA distinguishes three different models. The first one is the *Computation Independent Model* or *CIM* describing the business system. The second model is the *Platform Independent Model*, *PIM*, that is a view of a system from a platform independent point of view. The last one is the *Platform Specific Model*, a.k.a. *PSM*, a view of a system that

[1] http://www.omg.org/mda/

combines the specifications of the PIM with the details that specify how that system uses a particular type of platform [5]. The MDA specification [6] uses the term platform to intend not only a specific operating system but also a language-based platform such as Java or Python, and even common development practices such as the creation of accessor methods for class attributes [4].

MDA has introduced the notion of automatic transformation, that is one of its key features. A transformation describes how a model defined in a source language (source model) can be transformed into one or more models in a target language (target model). As shown in the left side of Fig. 1, in an MDA approach two consecutive transformations are employed. First, the platform independent models are transformed into platform specific models containing a lot of information about the underlying platform. In a subsequent transformation step, system code may be derived from the PSM.

Within the same MDA initiative, the Model Driven Testing (MDT) approach for software testing has been also proposed. MDT applies to test modeling the same philosophy followed by MDA in system modeling, as it is shown in the right side Fig. 1. The platform independent test design model (PIT) can be obtained from the system design model defined at the PIM level [7]. It can be transformed into platform specific test design models (PSTs). PSTs may also be directly derived from the PSM models. At last, starting from the PST, the test design models can be transformed into executable test code [2].

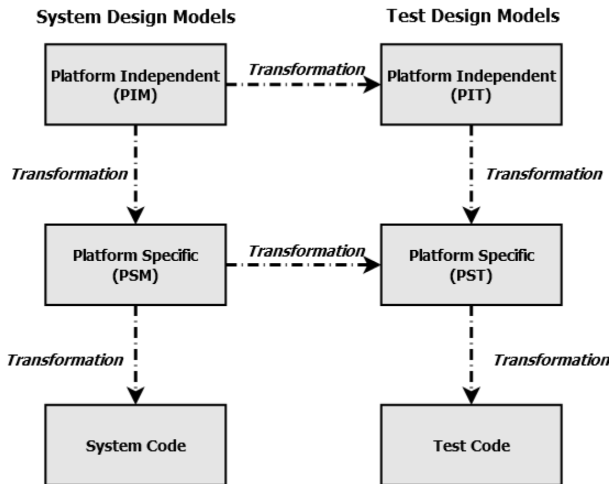


Fig. 1. Model Driven Architecture and Model Driven Testing

UML models play a fundamental role in Model Driven Architecture approaches and they are widely used for defining, at different levels, both static and dynamic aspects of the system. Among all the possible diagrams, we will focus on the finite state machine (FSM) ones (represented by UML StateMachine models), since they are appropriate to specify the behavior of event-driven systems, including Graphical User Interfaces (GUIs) and control systems. FSM are widely used as a basis for test generation and/or coverage analysis if they are specified with appropriate detail and rigor [8]. If we consider the MDA approach shown in 1 and we implement it using FSM models, the finite state machine at the PSM level will be transformed into a state machine at the PST level. The FSM at the PSM level will be in turn automatically transformed into a part of the source code of the whole system. If we switch to the MDT testing approach, we will have that the state machine at the PIT level will be transformed into the PST model and then the PST transformation will return the test code that exercises the generated source code of the system. Usually, the transformations from state machines at PST level towards test code are designed to guarantee that specific model coverage criteria are met, such as the coverage of all states, all transitions, all paths and so on [9].

III. Exploratory Study

In this section we present the exploratory study we performed for evaluating the differences that may exist between model and code coverage testing adequacy achieved by a test suite generated according to an MDT approach. More in detail, the study aimed at understanding:

- which levels of code coverage testing adequacy can be reached by a test suite generated to guarantee a specific model coverage testing adequacy?
- what are the factors that may cause the differences between code coverage and model coverage testing adequacy?

A. Study Process Description

The process we followed for conducting the study is shown in Fig. 2.

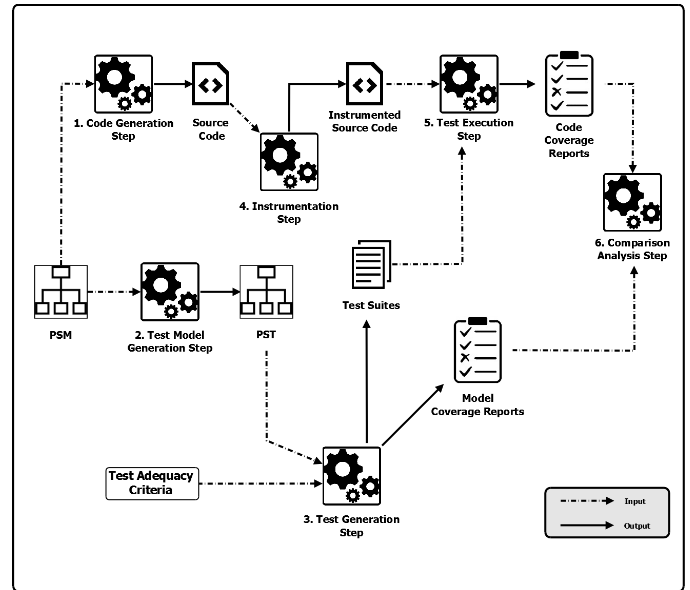


Fig. 2. Experimental Process

According to the MDA approach, in the *Code Generation* step, a UML StateMachine model [10] at PSM level is automatically translated into source code. On the other side, following the MDT approach, the PSM model is transformed into the corresponding UML StateMachine model at PST level by executing the *Test Model Generation* step. In the *Test Generation* step, the UML StateMachine model at PST level is used to automatically produce test suites able to meet a set of test adequacy criteria defined at model level. Furthermore,

this step generates output reports providing the model coverage reached by each test suite. In the *Test Execution* step, each test suite is used to test the auto-generated code. Before this step, the code must be instrumented to allow the evaluation of the test adequacy at code level. The output of this step is a report providing the code coverage testing adequacy obtained by exercising the auto-generated code by each test suite. Eventually, the *Comparison Analysis* step is executed for evaluating the differences between the model coverage and code coverage obtained by the test suites.

B. Metrics

Different metrics were exploited to conduct the study. To evaluate the complexity of the UML StateMachine models belonging to the PSM level, the following metrics were adopted:

- $\#States$: it represents the overall number of states of the model
- $\#Transitions$: it represents the overall number of transitions of the model
- $AACC$: it is the Average Action Cyclomatic Complexity. It is defined as the ratio between the sum of the cyclomatic complexity of all the actions of the model and the overall number of actions.

We proposed the latter metric for taking into account the different styles adopted by practitioners to design UML StateMachine diagrams. In the PSM model indeed it is possible to specify part of the system logic even by directly writing the code associated with an action.

For evaluating the model coverage reached by a test suite we exploited the following metrics:

- $CS\%$: it is the percentage of covered states
- $CT\%$: it is the percentage of covered transitions

Eventually, for measuring the code coverage obtained by a test suite we considered the metrics reported below:

- $CSTM\%$: it is the percentage of covered statements
- $CB\%$: it is the percentage of covered branches

C. Study Process Execution

As objects of the study we considered four different UML StateMachine models at the PSM level, named SM_1, SM_2, SM_3 and SM_4, having the complexity reported in Table I. These models describe the behavior of two simple real-world systems i.e., a garage management system (SM_1 and SM_2) and an order management system (SM_3 and SM_4). The process was executed four times, each time considering just one of these models.

TABLE I. MODELS COMPLEXITY METRICS

Metrics	SM_1	SM_2	SM_3	SM_4
$\#States$	4	4	8	8
$\#Transitions$	9	9	12	12
$AACC$	1	7	1	2.67

Each model was automatically translated into the corresponding Java source code (SC_1, SC_2, SC_3 and SC_4) by exploiting the Visual Paradigm CASE tool [2].

At the same time, the models at PSM level were manually transformed into the corresponding PST models having the same values of complexity metrics.

Each model at PST level was automatically translated into executable test suites exploiting the features provided by Conformiq Designer tool [3]. The test suites that were generated had to guarantee the following test adequacy criteria:

1) TA_1: it assures the coverage of all states
2) TA_2: it assures the coverage of all transitions

Two test suites were automatically generated for each model, overall eight test suites were produced. By analyzing the Model Coverage reports, we verified that all the test suites generated to guarantee the TA_1 criterion reached a value of $CS\%$ equal to 100%. In the same way, the remaining four test suites produced to meet the TA_2 criterion actually obtained a $CT\%$ equal to 100%. All test suites were ran against the auto-generated Java code previously instrumented with the CodeCover[4] tool.

The column 2 and column 3 of Table II report the results of code coverage reached by the test suites. A detailed discussion about the findings of these results is described in the next Session.

TABLE II. CODE TEST ADEQUACY REACHED BY TEST SUITES

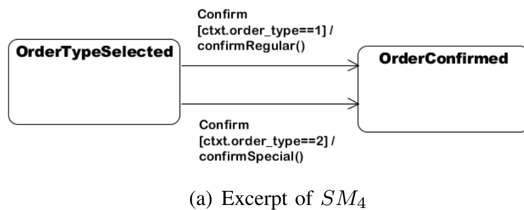
SC_1				
Test Suite	$CSTM\%$	$CB\%$	$CSTM\%$	$CB\%$
TestSuite(TA_1)	53.1 %	25.0 %	74 %	50 %
TestSuite(TA_2)	69.9 %	35.0 %	100 %	83.3 %
SC_2				
Test Suite	$CSTM\%$	$CB\%$	$CSTM\%$	$CB\%$
TestSuite(TA_1)	54.3 %	36.7 %	72 %	57.7 %
TestSuite(TA_2)	71.3 %	51.7 %	96.6 %	84.6 %
SC_3				
Test Suite	$CSTM\%$	$CB\%$	$CSTM\%$	$CB\%$
TestSuite(TA_1)	51.3 %	29.2 %	67.8 %	42.8 %
TestSuite(TA_2)	75.1 %	41.7 %	100 %	78.6 %
SC_4				
Test Suite	$CSTM\%$	$CB\%$	$CSTM\%$	$CB\%$
TestSuite(TA_1)	48.4 %	27.3 %	62.1 %	31.2 %
TestSuite(TA_2)	71.6 %	40.9 %	92.5 %	56.2 %

IV. FINDINGS

As data show all the executions of the test suites never reached the 100% of the $CSTM\%$ neither of the $CB\%$. This preliminary result demonstrates that there are differences between the model coverage and code coverage reached by executing the test suites. In order to understand which factors influenced these differences, we performed an analysis of

[2]http://www.visual-paradigm.com/
[3]https://www.conformiq.com/products/conformiq-designer
[4]http://codecover.org/

the Code Coverage reports. We observed that only a part of auto-generated code actually implemented the behavior of the StateMachine. The remaining code contained supporting logic and Exception Handling. The classes that actually contained the code related to the behavior of the StateMachines were `ControllerContext` and `Controller`. By further analyzing these classes we were able to recognize code that allows the application to deal with unexpected events. This is added by the Code Generator tool to guarantee the robustness of the produced software. Moreover, the auto-generated code contained statements related to the execution of the application in the *Debug mode*. This part of the auto-generated code cannot be covered, since the Test Generator does not take into account these unexpected events nor the execution in *Debug Mode*. We performed a further analysis where this additional code was not considered. The obtained refined results are reported in column 4 and column 5 of Table II. As data show, the test suites covering all the transitions always reached 100% of $CSTM\%$ of the code generated from models having $AACC$ equals to 1. The $CB\%$ values were always lower than 100%.



(a) Excerpt of SM_4

```
protected void Confirm(OMSControllerContext context)
{
    OMSController ctxt = context.getOwner();

    if (ctxt.getOrderType()==1)
    {
         (context.getState()).Exit(context);
         ctxt.confirmRegular();

         context.setState(OMSControllerFSM.OrderConfirmed);
         (context.getState()).Entry(context);
    }
    else if (ctxt.getOrderType()==2)
    {
        (context.getState()).Exit(context);
        ctxt.confirmSpecial();

        context.setState(OMSControllerFSM.OrderConfirmed);
        (context.getState()).Entry(context);
    } else
    {
        super.Confirm(context);
    }
    return;
}
```

(b) Event Code Coverage

```
public void confirmRegular() {

    total=total+(this.order_item*5);
}

public void confirmSpecial() {

    switch(this.order_item){
    case 1:{
        total=total+(this.order_item*10);
        break;
    }
    case 2:{
        total=total+(this.order_item*9);
        break;
    }
    case 3:{
        total=total+(this.order_item*8);
        break;
    }
    default:{
        total=total+(this.order_item*7);
        break;
        }
    }
}
```

(c) Actions Code Coverage

Fig. 3. Code Coverage Examples

A deeper analysis showed us that test suites covering all transitions on the model are not able to exercise all branches on the code. As an example Fig. 3 shows an excerpt of SM_4 and the code related to the events and actions of the two transitions. Moreover, Fig. 3(b) and Fig. 3(c) highlight the code exercised by executing a test suite able to cover the two transitions of the model. By analyzing the code coverage reported in Fig. 3(b), we were able to understand that the test suite covered the `(if (ctxt.getOrderType()==1)` decision but only the $TRUE$ branch of the `if (ctxt.getOrderType()==2)` one.

Moreover, the test suites were never able to reach 100% of both $CSTM\%$ and $CB\%$ of the code generated from models having an $AACC$ value greater than one. By analyzing the Code Coverage Reports we comprehended that test suites were able to execute all the actions but were not able to exercise all their code. As an example, Fig. 3(c) shows that, despite the two transitions were covered, the code related to the `confirmSpecial()` action, with a cyclomatic complexity equals to four, was not completely exercised. The code related to the `confirmRegular()` action were completely covered since it had a cyclomatic complexity value equals to one.

At the end of this exploratory study we were able to understand that the main factors influencing differences in model and code coverage of test cases automatically produced in MDA-MDT approaches are (1) the tool implementing the transformation rules from *PSM* towards system code, (2) the test adequacy criteria at model level exploited for the test code generation, and (3) the style adopted by the modeler to design the system behavior at *PSM* level.

V. Conclusions and Future Works

In this paper we investigated the existing differences between test adequacy at model level and code level reached by test suites automatically generated in the context of model driven approaches. To this aim we performed an exploratory study that allowed us to identify this differences and to understand the main factor influencing them.

The result of the study showed us that: (1) there are differences between model and code coverage and (2) the three main factors that may influence these differences.

As a future works we intend to propose a conceptual framework for the comparison between code coverage and model coverage in the context of model driven approaches. Furthermore this framework will be exploited to perform an empirical study involving a meaningful number of UML StateMachines, adopting different PSM to code transformation rules and considering several technologies and modeling styles.

References

[1] D. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb 2006.

[2] J. Zander, Z. Dai, I. Schieferdecker, and G. Din, "From u2tp models to executable tests with ttcn-3 - an approach to model driven testing -," in *Testing of Communicating Systems*, ser. Lecture Notes in Computer Science, F. Khendek and R. Dssouli, Eds. Springer Berlin Heidelberg, 2005, vol. 3502, pp. 289–303.

[3] A. Baresel, M. Conrad, S. Sadeghipour, and J. Wegener, "The Interplay between Model Coverage and Code Coverage," in *Conference On Computer Aided Systems Theory*, 2003.

[4] T. Meservy and K. Fenstermacher, "Transforming software development: an mda road map," *Computer*, vol. 38, no. 9, pp. 52–58, Sept 2005.

[5] N. Debnath, M. Leonardi, M. Mauco, G. Montejano, and D. Riesco, "Improving model driven architecture with requirements models," in *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, April 2008, pp. 21–26.

[6] "MDA Specifications," http://www.omg.org/mda/specs.htm/, 2015, [Online; accessed 19-July-2015].

[7] Z. R. Dai, "Model-driven testing with uml 2.0," Computing Laboratory, University of Kent, Tech. Rep., 2004.

[8] R. D. F. Ferreira, J. a. P. Faria, and A. C. R. Paiva, "Test coverage analysis of uml state machines," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 284–289.

[9] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[10] "UML Specifications," http://www.omg.org/spec/UML/2.5/, 2015, [Online; accessed 19-July-2015].