Shall We Test Service-Based Models or Generated Code?

Jean-Marie Mottu, Pascal Andre, Maxence Coutant, Tom Le Berre

Shall we test Service-based Models or Generated Code?

Jean-Marie Mottu
LS2N, Université de Nantes, IMT Atlantique
Nantes, France
jean-marie.mottu@ls2n.fr

Pascal André
LS2N, Université de Nantes
Nantes, France
pascal.andre@ls2n.fr

Maxence Coutand
Université de Nantes
Nantes, France
maxence.coutand@etu.univ-nantes.fr

Tom Le Berre
Université de Nantes
Nantes, France
tom.le-berre@etu.univ-nantes.fr

***Abstract*—Testing early reduces the cost of the validation process. Model-Driven Testing promotes the creation of the tests at the model level and their transformation into code which can be executed. However, we miss an empirical evaluation on the efficiency of generating tests designed on the service-based component models compared to implementing them based on the generated code.**

In this paper, we evaluate if one shall generate tests from platform-independent models or implement them on the generated code. We consider service-based component models and a case study modeling and generating the code of a platoon of vehicles. Our experimentation on that first case study confirms that generating tests from the service-based models is easier compared to implementing them on the generated code. Moreover, efficiency of the tests of both approaches is similar, considering code coverage and mutation analysis.

***Index Terms*—Key words : Model Driven Development, Model Driven Testing, Component, Test Harness, Correctness**

I. Introduction

Testing early reduces the cost of the validation process. Studies have shown that the most costly errors are those related to the definition of needs and not those related to code. This is particularly true in Model Driven Engineering (MDE), where the models should be validated before being transformed, preventing the spread of bugs introduced into the design. Indeed, even if the transformations are correct, the models could have bugs to be corrected before being transformed. We then create tests at the model level and following Model-Driven Testing (MDT) principles, they are transformed into code and executed on the generated code of the system under test (SUT).

We consider Service-based Component models and focus on test at three phases: unit, integration and system testing. Non-functional testing is outside the scope of this study depending on implementation languages and frameworks, but functional errors are detectable at the model level.

Our study focuses on the comparison between the test generated from model level and the test implemented on the generated code. Some research has achieved to propose MDT methods and tools, such as [1], or our COSTOTest platform [2] dedicated to service-based component models. They bring the benefits of MDE into testing which is beneficial considering several issues, such as:

- Heterogeneity of the code: when developing layered applications (client-server development, web, n-tiers...) it is difficult to implement test cases according to the layers.
- Separation of points of view and concerns: in the final code, business logic is melted with code and presentation. Separate their testing is more complicated than on the model.
- Domain expert can contribute easier to testing considering abstract models instead of generated code.

However, we miss an empirical evaluation on the efficiency of generating tests designed on the service-based component models compared to implementing them based on the generated code.

If it seems obvious that generate tests from service-based models is easier, we would like to confirm it experimentally. Moreover, the efficiency of such tests should be controlled to prevent that tests created on the models are not sufficient requiring additional effort based on the generated code.

To evaluate if one shall generate tests from service-based models or implement them on the generated code, we consider four research questions:

- *RQ1:* Is the coverage similar with tests from the two approaches?
- *RQ2:* Is the fault detection effectiveness equivalent between the two approaches?
- *RQ3:* Does the code generated from models require additional tests?
- *RQ4:* Is it simpler to create tests with one of the two approaches?

In this paper, we design an experimental process to answer the research questions. Our case study is a platoon system which models are defined by components and services via the Kmelia modeling language [3]. The interface of a component defines the services that it offers and those that it requires. Services exchange data through

communication channels that create a link between them, a dependency. The COSTO tool (COmponent Study Toolbox) allows to transform the system model defined in Kmelia into generated code of the SUT, implemented in Java. To test the model at code level we use COSTOTest tool [2] that allows us to create a test harness that we use to isolate the service to be tested and to generate executable tests.

The novelty of the current paper is an experimental protocol allowing us to get results on a first case study. They confirm that generating tests from the service-based models is easier compared to implementing them on the generated code. This latter contains platform dependent code which hides code dedicated to the modelised SUT. Moreover it doesn't offer testing facilities such as mock generation. We have also experimented that coverage and fault-detection effectiveness are equivalent in the two approaches.

The article is organized as follows. The Model-Driven Testing approach and how we can apply it on a case study thanks to COSTOTest are explained in this section II. The experimentation is detailed in section III. We discuss the results in section IV. The section V presents the related work. The section VI summarizes the results and introduces perspectives.

II. Model Driven Testing

In this paper we focus on the MDT approach when developing service-based components. In this section, we first remind the MDT process and use it to introduce both approaches considered in this paper: generation of the tests from the service-based models or tests implementation on the generated code. We then discuss in more detail the concepts of service-based component models considering the *platoon* case study as it is implemented with COSTO and tested with COSTOTest.

A. Model Driven Testing

Fig. 1 illustrates the Model Driven Development (MDD) on the right and the Model Driven Testing approach on the left. In the middle column of the Fig. 1, solid line rectangles illustrate the inputs of both approaches:

- System model which is a platform independent model (PIM) describes the SUT.
- Test Intentions (TI) declare which model's elements are tested, in which context, with which data. They model the test cases, based on the PIM.
- Operational Framework is a platform description model (PDM) describing how to transform the test harness in Java code for instance.
- Data sources provide input data and expected results (oracle).

On the left column of the Fig. 1, the Model Driven Testing process is made of three activities, implemented by model transformations. The first one builds the test harness as an assembly of the PIM with the TI and returning a Test Specific Model (TSM) (intermediate elements

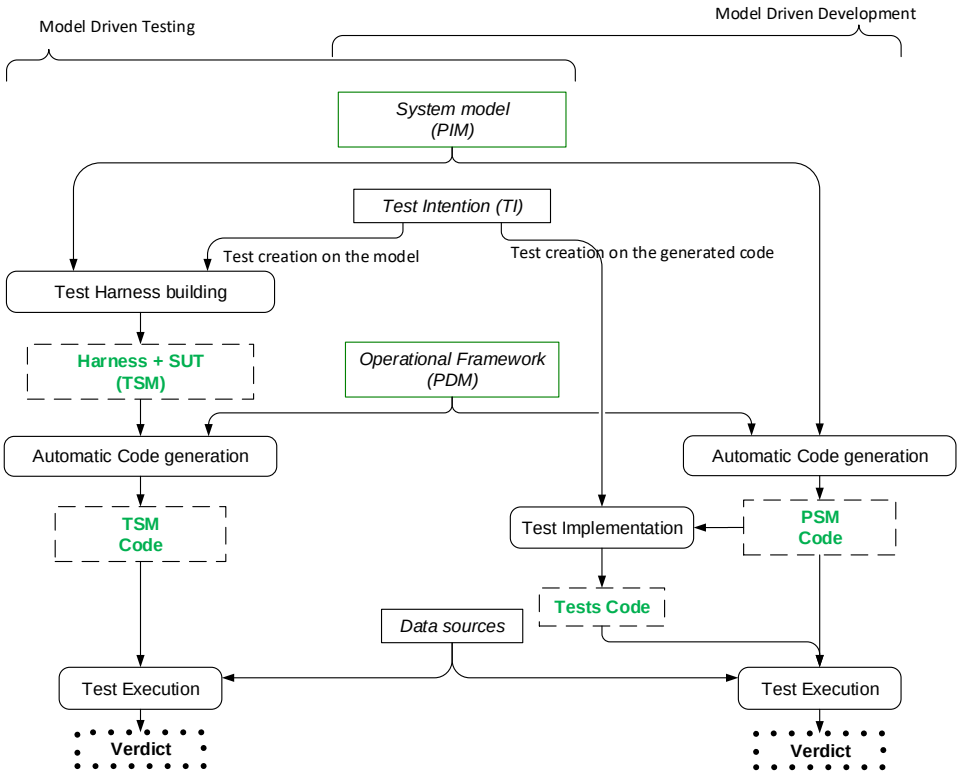

Fig. 1: Overview of the global process: Model Driven Testing on the left, Model Driven Development on the right

are in dashed line rectangles). The second transformation composes the harness with a PDM to get the executable code of both the SUT and its tests. During the execution, the data sources provide the test data needed to run the tests, and concrete assert functions return the verdicts (final results are in dotted line rectangles).

On the right column of the Fig. 1, the Model Driven Development process is also producing executable code of the SUT but the tests are not generated at the model level but implemented based on the PSM code and the Test Intentions.

In both MDD and MDT approaches we obtain an executable code of the system based on a PIM and model transformations. Additionally, MDT generates the tests and their harness when MDD focus only on the system generation and let the tester implements the tests on the system code. MDT is different than Model-Based Testing (MBT) which does not follow a model-driven architecture process. MBT can not consider the system and test harness generation based on a PIM and not take advantage of model transformations to platform specific code.

To answer the research questions, we will experimentally compare those two approaches:

- RQ1 and RQ2: by considering the TSM Code compared to the Tests Code and PSM Code.
- RQ3: by considering the Automatic Code Generation and Test Implementation.
- RQ4: by considering the Test Harness Building compared to the Test Implementation.

B. Service-based component models

In this paper we continue our work of [4], considering service-based component models, such as the one illustrated Fig. 2. We recall the main features of the modelling language. A component system is an assembly of components where the services required by a component are bound to provided services by the means of assembly links. The interface of a component defines its provided and required services. The interface of a service defines a contract to be satisfied when calling it. The services may communicate, and the assembly links denote communication channels. The set of all the services needed by a service is called its service dependency. The required services can then be bound to provided services. These needs are either satisfied internally by other services of the same component, or specified as required services in the component's interface and satisfied by other components. A composite component encapsulates an assembly.

Our case study is illustrated in Fig. 2: a simplified platoon of several vehicles, as used in [2]. Each vehicle computes its own state (speed and position) by considering its current state, its predecessor's state, and also a safety distance between vehicles. Each vehicle is a component providing its speed and position and requiring predecessor's speed and position. Three vehicles are assembled in that example. Each vehicle provides a configuration service conf initiating its state, a service run launching the platoon and requiring the service computeSpeed to calculate new position and speed. The leader is another component controlling its own values according to a position goal. The Fig. 2 represents completely only the dependencies of computeSpeed, the service under test considered in the experimentation.

C. Test harness creation and code generation

The process consists of designing test cases for the architecture of components, then executing them to obtain verdicts. Usually the tester works with a test intention that gives him/her the specification of test cases: model's elements under test, in which situation, with which input data and oracle. This abstract intention is embodied throughout the process to finally give a test harness according to a process detailed in [2].

We have developed COSTO a set of prototype tools dedicated to design, transform and execute Kmelia service-based component models [3]. In particular, we have developed COSTOTest [2], a COSTO plugin to implement Model Driven Testing on service-based component models.

During the step *Test Harness Building*, COSTOTest is used to assist the tester, by helping the mapping between the test intention and the system under test. As illustrated

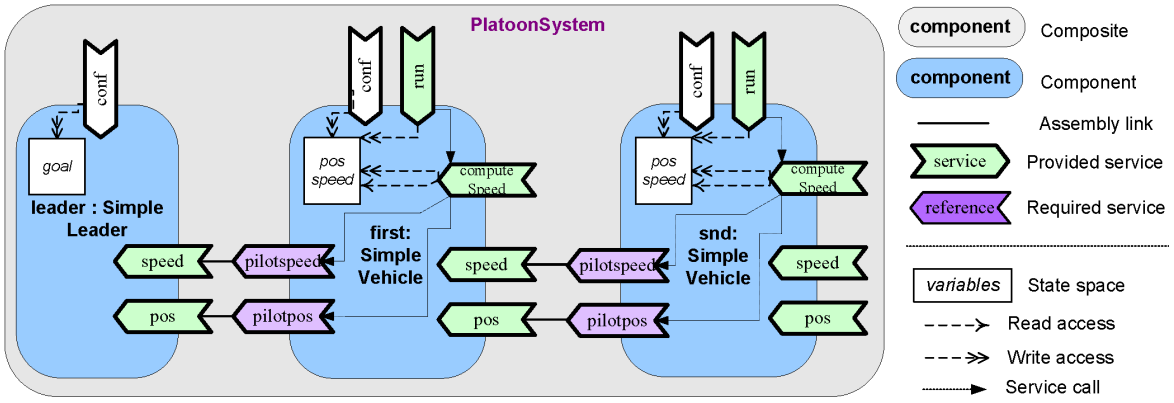

Fig. 2: Kmelia components and services for the Platoon system

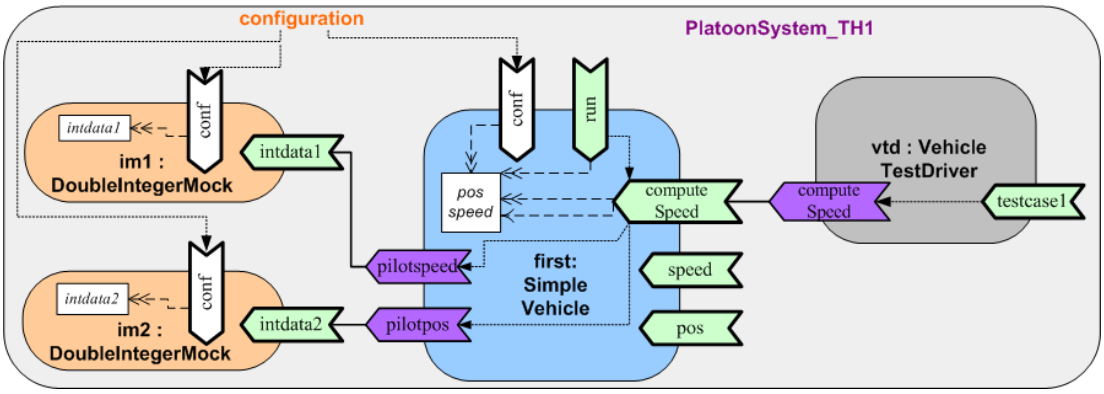

Fig. 3: Test harness for the computeSpeed service of the component SimpleVehicle under test [2]

Fig. 3, the tool produces mocks and drivers (im1 and vtd, respectively), which are connected to service under test (computeSpeed service) and to services requiring test data (pilotspeed and pilotpos).

Once the model of the test harness is built, its TSM code can be generated in Java and concrete data mapping is performed by matching the data sources on the entry points present in the harness, as explained in [4].

III. Experimentation

In this section, we detail the protocol of the experimentation and the experimental results obtained from the *platoon* case study[1]. Each subsection considers a research question, in their respective order.

A. Model and Code Coverage (RQ1)

We compare the coverage of the two approaches. First, we build the Test Intentions creating a test data set based on the coverage of the model of the PIM[2]. Second, we build the Test Harness and generate the TSM Code to measure how the generated tests cover the TSM code. Third, we implement the Tests Code based on the Test Intentions and the PSM Code generated by the MDD approach to measure how the Tests Code cover the PSM code. Finally we compare the respective test coverage of both approaches.

1) Data: The test data set is an essential element for the tester for code coverage. We have the objective of covering all the transitions of the state transition system of the service that is modeled in the PIM. We are studying the computeSpeed service in the experimentation. As shown by the graph of Fig. 4, there are nine possible paths in the state transition system associated with the computeSpeed service. However, there are three paths that cannot be covered because the value of some variables changes during the transition and it is mathematically impossible to cover certain branches later on:

- e1_e2_8 && e2_f_11 because for the transition e2_f_11 it is necessary that newspeed < 0 or e1_e2_8 implies that newspeed = vspeed + distanceStep. However, we have vspeed >=0 && distanceStep > 0, newSpeed>0.
- e1_e2_10 && e2_f_11 because for the transition e2_f_11 it is necessary that newSpeed < 0 while e1_e2_10 implies newSpeed = 0
- e1_e2_10 && e2_f_12 because for the transition e2_f_12 it is necessary that newSpeed > maxSpeed. maxSpeed >= 0 inherently and e1_e2_10 implies newSpeed = 0, thus maxSpeed >= newSpeed.

We therefore use six test cases to cover all the branches of the automaton associated with the computeSpeed service.

2) Coverage: The computeSpeed service is transformed into two java methods `computeSpeed` and `computeSpeedLTS`. Table I shows the graph node coverage of these test cases where CS stands for `computeSpeed` and CSLTS stands for `computeSpeed_LTS`.

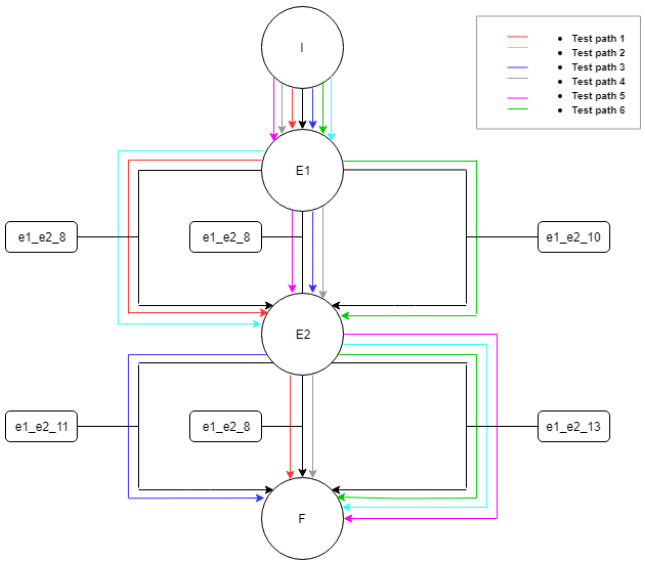

Fig. 4: Control graph of the computeSpeed service

[1]The experimentation models and source code are available at https://costo.univ-nantes.fr/download/

[2]Capitalized names are illustrated in Fig. 1

TABLE I: Node coverage by the 6 test cases

M	TC1	TC 2	TC 3	TC 4	TC 5	TC 6
CS	78.9%	81.0%	79.5%	78.9%	81.0%	77.5%
CSLTS	95.5%	95.3%	93.8%	94.5%	95.3%	95.3%

Code coverage is a measure that can be important in describing the source code rate executed by a program when a test suite is started. A program with high code coverage, measured as a percentage, is considered to have more code executed during testing, suggesting that it is less likely to contain undetected software bugs, compared to a program with low code coverage. There are different metrics for calculating code coverage. We use the instruction coverage as well as the branch code coverage.

The coverage analysis is processed with the *EclEmma* eclipse plugin[3] which is based on *Jacoco* code coverage libraries. It provides us with information on the code coverage for which we used two metrics:

- Coverage Instruction (CI) : measures the percentage of line of code executed and verified.
- Coverage branches (CB) : measures the percentage of branch executed and verified.

TABLE II shows the coverage of the code generated by the harness with all test cases allowing to have all the branches of the automaton covered. We notice that the code coverage of both approaches is similar. COSTO generates computeSpeed and computeSpeedLTS similar enough in both the TSM code and PSM code to get the same code coverage.

TABLE II: Coverage of the 6 cumulative tests

method	TSM code		PSM code	
	CI	CB	CI	CB
computeSpeed	89.9 %	85.7 %	89.9 %	85.7 %
computeSpeedLTS	98.4 %	92.9 %	98.4 %	92.9 %

3) Origin of the not covered code: We have created the test data set to fully cover the transitions and states of the automaton associated with the computeSpeed service. However, we do not obtain 100% coverage of instructions and branches, so we analyze the origin of this non-covered code. We identify several situations:

- In the case of `if ( transition ==null) return`, only one of the two branches is covered. That instruction aims to avoid bug cases where the "transition" field has not been filled in. However, these branches are not part of the basic transitions of the model. Therefore, they are not to be taken into account in calculating the coverage of our service.
- The case of `return true` corresponds to the code generated to satisfy the needs of the language. Some functions being composed of a succession of ifs returning a Boolean without having any other, the function must necessarily return some outside the latter.
- In the case of *partially covered if statement (1 branch out of 2)*, the part of the not covered branch corresponds to an error case that will never happen because it is not a branch of the model but a branch created to satisfy the needs of the language.
- In the case of *pre- and post-conditions*, a Kmelia feature, the functions corresponding to the verification of the preconditions generated by the framework are never called when the framework configuration toggles off the assertion control. Indeed, the preconditions are considered as being in defensive mode, this implies that for the framework all the preconditions are respected and does not call consequently the functions allowing this verification.

Once the previous cases are removed from our code coverage calculation, we get 100% coverage, which is consistent with the data set we used.

B. Mutation Analysis (RQ2)

Code coverage is not a sufficient criteria to evaluate the quality of test cases [5]. Indeed, the tests can obtain good coverage rates but still not detect every error. To overcome this problem, it is recommended to use mutation analysis. Artificial bugs (called mutation) are introduced into the program, creating mutants, with one mutation each. Once the mutations have been injected into the program, we run our tests. If the tests performed can distinguish the behaviour of the mutants from that of the original program (i.e. the test has failed on a mutant but has passed on the original program, or the contrary), they are considered to be of good quality. If the tests performed do not detect the presence of a mutation in the mutant, then it means that the current tests are not sufficient to ensure the reliability of the program. Developers have to create additional tests.

In the same way than when considering the coverage, we apply the mutation analysis on the two approaches and compare their respective results. Considering MDT, mutations are applied on the PIM. Considering MDD, mutations are also applied on the PSM Code generated.

We use three kinds of mutation operators: Finite-State Machine operators [6] are used to inject mutations in the automata modeling the behavior of the service (Fig. 4). Relational and arithmetic mutation operators are applied to inject mutations in the expressions.

Finite-State Machine operators are applied differently considering MDT and MDD. Considering MDT, they are applied at the model level by modifying the Kmelia code of the PIM. These mutations introduce a change in the structure of the original system. For instance, in the case of computeSpeed, the transitions going to state E2 are changed to go to state F. It produces 4 viable mutants (that still satisfy well-formedness rules and can be transformed). Considering MDD, these mutations request to modify several functions in the two classes computeSpeed and computeSpeedLTS.

[3]https://www.eclemma.org/jacoco/

	==		>=		>		<		<=		!=	
INSTRUCTIONS ALLANT SUBIR MUTATION	Killed	Alive	Killed	Alive	Killed	Alive	Killed	Alive	Killed	Alive	Killed	Alive
[distance>safeDistance && vspeed<=pspeed]	2	4	0	6			2	4	3	3	1	5
[distance>safeDistance && vspeed<=pspeed]	2	4	4	2	2	4	0	6			1	5
[distance>safeDistance && vspeed>pspeed]	3	3	0	6			3	3	3	3	1	6
[distance>safeDistance && vspeed>pspeed]	3	3	0	6			5	1	4	2	1	5
[distance <=safeDistance]	1	5	2	4	0	3	0	6			1	5
[newspeed<0]	1	5	3	3	3	3			0	6	1	5
[newspeed>maxSpeed]	2	4	0	6			3	3	3	3	1	4
[newspeed<=maxSpeed && newspeed>=0]	3	3	2	4	3	3	1	5			1	6
[newspeed<=maxSpeed && newspeed>=0]	2	4			1	5	3	3	3	3	2	4

Fig. 5: Relational Operator Replacement Mutator (ROR)

	+		-		*		/		%	
	Killed	Alive	Killed	Alive	Killed	Alive	Killed	Alive	Killed	Alive
newspeed:=vspeed+distanceStep			2	4	0	6	2	4	2	4
newspeed:= vspeed - distanceStep	2	4			2	4	2	4	3	3

Fig. 6: Arithmetic Operator Replacement Mutator (AOR)

TABLE III: AOR and ROR mutations

Original	AOR1	AOR2	AOR3	AOR4
+	-	*	/	%
-	+	*	/	%
*	/	%	+	-
/	*	%	+	-
%	*	/	+	-

Original	ROR1	ROR2	ROR3	ROR4	ROR5
<	<=	>	>=	==	!=
<=	<	>	>=	==	!=
>	<	<=	>=	==	!=
>=	<	<=	>	==	!=
==	<	<=	>	>=	!=
!=	<	<=	>	>=	==

In Table III we list the mutations applied on each arithmetic and relational operator found in the program.

Considering MDT, we apply these mutation operators on the PIM, then generate the TSM Code for each mutant. Considering MDD, we use the tool PIT-Test[4]. This tool creates automatically AOR and ROR mutants, to execute the tests, then to return a detailed verdict. Below, the two tables summarize the results of the tests following the insertion of the AOR and ROR mutants. All the mutants with finite-state machine mutations have been killed.

The mutation scores and so the tables are the same considering MDT and MDD approaches. The results are: (i) 57 mutants generated (ii) 9 alive (iii) 84.21% of mutants killed (iv) 15.79% of mutants alive The percentage of mutants killed is 84.21%, which means that the data set is relevant for the computeSpeed service. In Fig. 7, the PIT-Test coverage results show that the test cases allow for very early coverage and that mutants are well detected. The number of mutants created and killed is different than the previous enumeration because PIT-Test introduced mutations in part of the generated code which is technical code not coming from the PIM but the PDM and then not useful for us for this experiment.

After repeating the operation on both types of test approach, we obtain exactly the same results. The mutation

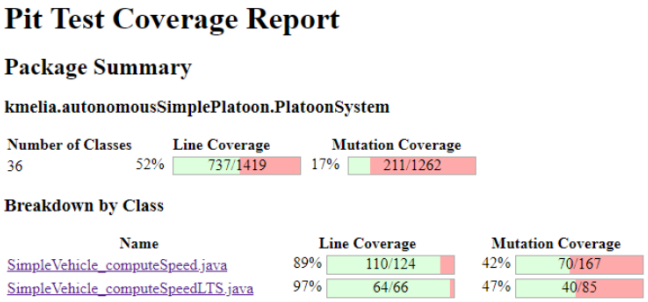

Fig. 7: PIT Test Coverage Report

[4]http://pitest.org/

analysis on the PSM code and the mutation analysis on the TSM code (created at model level) allow us to deduce that considering this experimentation the quality of the two test approaches is similar.

C. Additional Tests at the code level (RQ3)

Since the service under test is transformed from the PIM to the TSM Code, it may need additional tests to the ones created at the model level. Considering this, we should take care of the granularity and the separation of concerns.

A test is created based on a test intention: which model's element is tested, in which context, with which data. At the model level, one can create *unit tests* focusing on one service of one component. The granularity is different at the TSM Code level, since model elements are transformed into several code expressions with dependencies with different part of the generated code. Therefore, the granularity is already *integration test* at the code level because one service of one component involves considering several classes and methods. Therefore, those dependencies have to be considered and we check if they request additional test during the experimentation. Thanks to the MDT

framework this is not the case since it is able to manage the creation of mocks (as illustrated Fig. 3). A TSM mock is a component including communications facilities. Its code is generated and assuming the framework has been validated, we can be confident in its quality.

The separation of concerns is a benefit of using model-driven approaches. Creating the PIM, one can focus on the specification of the SUT without considering implementation and deployment of the system, which is managed by the PDM. Merging the TSM with the PDM, dilutes the testing concerns in the code of the TSM. For each method that overrides a PDM method, a recurrent question was shall we test it or not despite we *assume the PDM tests being tested apart from the generated code testing.*

For example, we implement manually the Test Case 1 of TABLE I by calling PDM primitives. It appears clearly that this test method is not trivial to write (and even to read). Without guidelines, the tester does not really know which method to test and how to configure the test.

Listing 1: Testcase1 - PSM Code testing

```
private static void testCas1(PlatoonSystem comp){
  comp.getMid().setConfig("conf", "mid", 100, 130);
  comp.getLast().setConfig("conf", "last", 27, 121);
  //creation of vehicle and driver services
  comp.initBindings();
  //creation of vehicle and driver services
  comp.getMid().createServices();
  comp.getLast().createServices();
  comp.getDriver().createServices();
  //initialisation of the threads
  comp.getMid().init();
  comp.getLast().init();
  comp.getDriver().init();
  Integer safeDist = 72;
  // service call
  ((SimpleVehicle_run) comp.getLast().getProvidedService("run")).
    callService("_computeSpeed", "computeSpeed",
      new Object[] {safeDist}, (SimpleVehicle_run)
          comp.getLast().getProvidedService("run"));
  Object[] rcvresult = ((SimpleVehicle_run) comp.getLast().
    getProvidedService("run")).receiveServiceReturn("_compute
    Speed", "computeSpeed", new Class<?>[] {Integer.class},
    (SimpleVehicle_run)comp.getLast().getProvidedService("run"));
  Integer newspeed = (Integer) rcvresult[0];
  //result for test case 1
  assert(newspeed ==130);
}
```

The examples of section III-A3 shows that the PDM interfere with code testing despite the fact that the code generated by COSTO has a great proximity with the model concepts: every model element is being traceable at the implementation level.

D. The complexity of implementing tests (RQ4)

During the first steps of the experimentation, we observe that it was easier for the students to test at the model level because they had no experience on the PDM. Then they proceeded by trial and error until having a sufficient knowledge to understand the missing cases of section III-A3. Many questions arise. Which classes have to be tested? Shall we retrieve the SbC model structure (how to isolate the 'components' and 'services'?)? How to configure the services and data inputs? Can we pass through the communications? Elements of answers are discussed in this section.

In the next two sections we will detail and compare the implementation of tests in each of the two approaches.

1) Design of the TSM and Generation of TSM Code: the Kmelia model of Fig. 2 is quite simple. It includes 3 components, but only 2 component types: SimpleDriver and SimpleVehicle. SimpleDriver defines one variable (the goal) and provides conf, pos, speed, goalreached, giveSafeDistance public services. All of these services have simple sequential state transitions. SimpleVehicle defines four variables, provides seven public services conf, run, pos, speed, goalreached, computeSpeed, giveSafeDistance and one private service applySpeed and requires pilotpos, pilotspeed, isgoalreached, safeDistance services. Only the run service includes a more complex behaviour with cycles. All of these services have simple sequential state transitions. We therefore have thirteen sets of tests to set up to test individually and functionally all the services of our system at the model level (and not only computeSpeed service).

In the example of Fig. 3, the TSM built for the computeSpeed service under test (SUT) involves also three components: the base SimpleVehicle, one mock component IntegerMock instantiated twice and a generated tester component TESTER_PlatoonTestIntention. The model metrics in TABLE IV show a low complexity.

TABLE IV: Metrics Comparison between Models

Metrics	PIM	TSM
Number of component types	2	3
Number of services	14	11
Number of Assembly links	8	3
Max LTS Complexity	O(n)	3

The resulting TSM is a first class (test) model: it has its own life cycle and the same PIM tooling is applicable (verifications and transformations). There is no need to replay the test harness building to refactor it by transformation for instance. It is then possible to add new test intention, to replace a Mock by another one for instance with the same facility than considering JUnit test cases.

During the creation of the TSM, the COSTO tool provides assistance to the tester to find the missing elements: unbind services, candidate services, missing variable configuration, missing test intention variables, type mismatches... All this information, that helps to get consistent and complete tests cases, is not so easily available at the code level. The COSTO tool automatically generates the Java code of the TSM. Thus, the respect of the initialization and execution process of the tests is no more the responsibility of the tester, which prevents human mistakes.

2) Implementation of tests in the code approach: the Kmelia specification includes libraries of constants, types

TABLE V: Metrics Comparison between Java Applications

Metrics	PDM framework				PSM generated code				TSM generated code			
	Total	Mean	Std. Dev.	Max	Total	Mean	Std. Dev.	Max	Total	Mean	Std. Dev.	Max
NbC	55	6,111	5,587	16	36				36			
NbA	139	2,527	3,474	17	68	1,889	1,39	7	74	2,056	2,068	11
NbM	540	9,818	12,012	57	389	10,806	8,95	47	388	10,778	9,363	47
LOC	3530				2360				2465			
CC		1,358	0,972	12		1,362	1,451	15		1,395	1,534	15

NbC: Number of classes (avg/max per package) / NbA: Number of attributes (avg/max per type) / NbC: Number of methods (avg/max per type) / LOC : Total lines of code / CC : Mc Cabe Cyclomatic Complexity (avg/max per method)

and functions (PLATOONLIB for the base application and PLATOONTESTLIB for the harness. These primitive elements are implemented in the target framework (*e.g.* java) by a mapping configuration file. In particular, the data values are provided by a specific access primitive.

In Table V, we compare the PDM java code, the generated PSM code and the System and TSM code (*cf.* figure 1) according to various metrics[5].

The value of PSM and TSM metrics are close, because the model SimplePlatoonSystem is itself a reduction of the base PlatoonSystem case study. In ordinary cases, the test harness is much simpler than the full application.

Despite the case is quite simple, the number of class of the generated PSM application is important. This single metrics shows that testing the model will be simpler than testing the code (or even model based testing if a model information is used to test the code). In the PDM (from which depend the PSM and TSM), the number of classes is only a part of the concurrency complexity since each service instance is a concurrent process that shares the component resources. The communications are performed through message buffers with asynchronous communications but synchronous calls or returns.

The generation of the code necessarily implies an addition of technical elements for the proper functioning of the program. The generation for the PSM code created 36 classes including several utility classes (one enumeration type, one class for the primitive types library mappings, a main program), a system configuration class, 11 classes for the driver component and 21 classes for the vehicle component and related services. The utility classes are not in the scope of the test bed (done once before). So we have 33 classes left to analyse and potentially test. Table VI give the metrics for the computeSpeed service, implemented by two java classes, one for the service and one for the behaviour (LTS) which complexity is higher. In this case one service refines into 52 methods. Again this single metrics shows that testing the model will be simpler than testing the code.

It must be taken into account that most of the generated code allows the system to operate according to the framework principle. So, for testing to be really useful, it

[5]Metrics 1.3.6 available at http://metrics.sourceforge.net/.

TABLE VI: Metrics Comparison between Service Code

Metrics	computeSpeed			computeSpeedLTS		
	Mean	Std.	Max	Mean	Std.	Max
NbA	7			1		
NbM	47			5		
LOC	224			80		
CC	1,064	0,32	3	6,6	6,859	15

is enough to test our code from a functional point of view. That is to say, independently testing each function of each class is not interesting. It is necessary to test if each service respects well what we want it to face. To do this, as with the model-level testing approach, we want to isolate each service from each other. To do this we also use mocks.

As Fig.1 shows, we are testing the generated *PSM code*. Thus, we must take into account the structure of the system to set up the tests. The implementation of mocks from this test approach is therefore more complex than the model-level test approach. Indeed, the only viable method we found to create the mocks of a service was to overload the classes managing the parent component of that service. In this way, we can modify the structure of the component and associate it with the mock we previously created. Creating a mock also means creating a new service, because it is a complete service that is doubled. Thus, there is a development effort to be made that is not present in the test approach at the model level. It is necessary to write 3 classes (the overloaded component and the two classes managing the mock service).

Then, you have to write the logical part of the test. That is to say, the part of the code that will allow the system to be configured, executed, and the test verdict rendered. The time required for this implementation can be long and may require iterating the launch and rewriting of tests in order to find the right configuration and order of the different components involved in the test. As stated in [4], *"Finding the services to invoke to put the components in an acceptable state for testing, is therefore not trivial."*

IV. Discussion

We found that the code coverage analyses from the model-based testing approach and the code testing ap-

proach yielded the same results. The percentage of code covered being the same implies that the tests from the model approach are at least as efficient as those performed by the code approach. This statement was confirmed by the mutation analysis carried out because we got the same mutation scores. However, there is a difference between these two approaches in the complexity of test implementation. Indeed, our study shows us that it is more difficult to implement tests using the code approach than the model approach. This difficulty is mainly expressed by the time spent analyzing and implementing the tests. In particular, as we have seen in the section on test implementation, in the model-based approach to testing we have the possibility to forget the structure of the system. This simplifies the implementation of mocks compared to the approach to testing at the code level.

A relevant question is also whether the tests from the level-model approach were more, or less, sensitive to system changes compared to the tests from the code-level approach. Both approaches treat the test input data in the same way. The data is specified in an independent XML file. In addition to allowing portability and simple reuse of test sets, it also allows them to be easily modified. So we can't compare them on how they process the input data. From the point of view of the evolution of the tests and their maintainability, the difference between the two test approaches could be in the amount of information that will have to be modified in the event of an evolution of the system under test. On this point, the test approach at the model level, freeing itself from the technical constraints of the final language and therefore being more abstract than the code, allows for better maintainability over time. As mention in [4]: "Defining tests at the model level makes it easier to adapt them if the model changes".

The adaptability of the tests of both approaches can also be based on their ability to evolve with their technical foundation. If the java base evolves, then the code will also have to evolve. An evolution of this type requires a phase of non-regression tests. The code testing approach involves manually modifying the tests each time the Java platform evolves to ensure that they meet the new standards, and that new errors have not occurred. The model-based approach to testing implies that test codes will be generated automatically as the PSM code. Thus, they will be automatically generated according to the current Java platform. It will therefore simply be sufficient to generate the tests again, then execute them. It is therefore clear that tests based on the model approach are more advantageous in the face of changes in the technical basis.

The PDM greatly influences the results of the experimentation. Having a distributed system middleware increases the distance between the model paradigms and the implementation paradigm. Distribution over internet devices (*e.g.* J2EE or .NET implementations) or the cloud will make it harder to test at the code level.

V. Related work

As far as we know, the topic of MDT is less addressed by researchers than model-based testing, except for UML Testing Profile (UTP)[6]. In their book [1] Baker and al. illustrates how UTP can be used for test modeling and test specification especially using frameworks like TTCN-3 and the JUnit test framework for Java. Our approach is less systematic in building the harnesses but provides a better level of automation when considering service-based component model. We have a vision similar to Born et al. who test UML component models deployed on Corba and test built-in contracts via TTCN-3 [7], [8]. However, we miss details about their implementation to compare precisely. In particular, the test code seems embodied in the model and not generated from test models.

Our approach is an answer to the research question on in Model-Driven Test-Case Construction raised in [9]. It allows the reuse of tests for different versions or variants of a system or for multiple systems within a family of systems and enables domain experts without programming skills to specify tests.

In [10], the author conducted an exploratory study to evaluate the differences that may exist between the model coverage guaranteed by the test scenarios and the code coverage achieved when they were run on the automatically generated code. They focus only on the code coverage, when we also consider the mutation analysis, the complementary tests, and the test creation effort. They have observed similar results: the coverage is similar between the two approaches after having selected the code not injected by the framework. However, when the code cyclomatic complexity was greater than 1, they observe differences. Then we have consider another service to check this issue with COSTOTest, but it remains similar when the cyclomatic complexity varies. Only the quality and number of test cases influence the coverage of the code and model. Thus, as the cyclomatic complexity increases, the number of test cases will have to be larger to fully cover the control flow graphs of the model, and the code.

Basically, Kmelia models are architectural models even if the specifier can go deeply in the service specification. In [11] Keum et al. also deals with complex communication middleware in the PDM. They focus on integration testing (architectural level) of service-oriented applications where the PDM is the SOAP standard. Our work can be seen as complementary because we enable unit and functional testing at a lower level. One important difference is that instead of testing the communications consistency, we verify them by the means of model checking and combine verification techniques to cover various kinds of properties [12]. We therefore address most of the issues given in the systematic literature review of Uzun et al. on Model-driven architecture based testing [13] except the analysis of results which is still manual in our case.

[6] http://utp.omg.org/

In [14], the authors propose a MDT approach for functional testing. According to the authors one of the main challenges for functional test case generation is the lack of formalism. They propose five meta-models and four transformations from requirements to test cases, that can be activity diagrams for example. This work does not execute the model of the tests but we think it is complementary to our process. Indeed it can be helpful to manage our test intentions which are currently seen as individuals in our approach.

In [15], Mussa et al. made a survey of 15 MDT tools and approaches. Most of them focus on the automatic generation of test cases from UML models. Even if the paper shows the benefit of MBT, it is not discussed experimentation having measured efficiency such as we do here. The survey [16] of Dias Neto et al. covers a great variety of MBT approaches that cover all the range from unit testing to system testing. The synthesis table shows a real complexity of testing steps, a variable level of automation with about two thirds of them having a tool support. Our approach covers until system testing without graphical user interfaces and non-functional testing. But it is clearly missing a test management level to handle the test intentions and also an continuous integrated process from requirements to test harness building and execution.

VI. Conclusion

In this article we have experimented and compared two test approaches. One is based on tests generated from the model level and the other on tests implemented on the generated code. First, this research shows us that the coverage rates, and the mutation analyses performed, are equivalent on both test approaches. We therefore conclude that the tests at the model level are as efficient as those performed at the code level. Second, we find that the complexity of implementing tests at the code level is higher than that based at the model level. In addition, it results from this work that model-level tests are more viable over time than code-level tests, because they are less affected by changes. Once the test harness of one service under test is built, a full series of tests, including non-functional ones, can be processed by modifying the test model. This approach allows a better separation of concerns helping model evolution.

Future developments could focus on tools to automatize the harness building because this activity is still time consuming. We currently work on a *Xtext* version of the test harness building that will enable to integrate better test intention management and test modelling. It would allow us to extend the experimentation with more and larger case studies. It would also be possible to consider the comparison of MDT with classic development (not even MDD) where the tests are created to cover the source code of the SUT.

Finally, in future work, we would like to experiment the coupling the model-driven testing approach with model-based testing. The aim is to automatically generate part of the test sets based on pre- and post- conditions.

References

[1] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag, Berlin, Heidelberg, 2007.

[2] Pascal André, Jean-Marie Mottu, and Gerson Sunyé. Costotest: a tool for building and running test harness for service-based component models. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2016.

[3] Pascal André, Gilles Ardourel, and Christian Attiogbé. Composing components with shared services in the kmelia model. In *International Conference on Software Composition*, pages 125–140. Springer, 2008.

[4] Pascal André, Jean-Marie Mottu, and Gilles Ardourel. Building test harness from service-based component models. In *proceedings of the Workshop MoDeVVa 2013 associated with Models2013 conference*, Miami, USA, October 2013.

[5] Alper Sen and Magdy S Abadir. Coverage metrics for verification of concurrent systemc designs using mutation testing. In *2010 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 75–81. IEEE, 2010.

[6] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the ISSRE'99*, pages 210–219, Washington, DC, USA, 1999. IEEE Computer Society.

[7] Marc Born, Ina Schieferdecker, Hans-gerhard Gross, and Pedro Santos. Model-driven development and testing - a case study. In *First European Workshop on MDA with Emphasis on Industrial Application*, pages 97–104. Twente Univ., 2004.

[8] Hans-Gerhard Gross. *Component-based Software Testing With Uml*. SpringerVerlag, 2004.

[9] Stefan Baerisch. Model-driven test-case construction. In *Proceedings of ESEC-FSE'07*, pages 587–590, New York, NY, USA, 2007. ACM.

[10] Domenico Amalfitano, Vincenzo De Simone, Anna Rita Fasolino, and Vincenzo Riccio. Comparing model coverage and code coverage in model driven testing: an exploratory study. In *30th IEEE/ACM ASE Workshops*, pages 70–73. IEEE, 2015.

[11] Changsup Keum, Sungwon Kang, and Myungchul Kim. Architecture-based testing of service-oriented applications in distributed systems. *Information and Software Technology*, 55(7):1212 – 1223, 2013.

[12] Pascal André, J. Christian Attiogbé, and Jean-Marie Mottu. Combining techniques to verify service-based components. In *Proceedings of the 5th International Conference, MODELSWARD*, pages 645–656, Porto, Portugal, 2017.

[13] Burak Uzun and Bedir Tekinerdogan. Model-driven architecture based testing: A systematic literature review. *Information and Software Technology*, 102:30 – 48, 2018.

[14] J.J. Gutiérrez, M.J. Escalona, and M. Mejías. A model-driven approach for functional test case generation. *Journal of Systems and Software*, 109:214 – 228, 2015.

[15] Mohamed Mussa, Samir Ouchani, Waseem Al Sammane, and Abdelwahab Hamou-Lhadj. A survey of model-driven testing techniques. In *2009 Ninth International Conference on Quality Software*, pages 167–172. IEEE, 2009.

[16] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the WEASELTech'07 WorkshopASE'07 2007*, pages 31–36, New York, NY, USA, 2007. ACM.