# Model Driven Architecture based Testing Tool based on Architecture Views

Burak Uzun and Bedir Tekinerdogan
*Information Technology Group, Wageningen University, Wageningen, The Netherlands*

Keywords: Model based Testing, Software Architecture, Architecture based Testing, Software Architecture Viewpoints.

Abstract: Model Driven Architecture Based Testing (MDABT) is a testing approach exploiting the knowledge in the design phase to test the software system. MBT can use different representations of the system to generate testing procedures for different aspects of the software systems. The overall objective of this paper is to present a model-driven architecture based testing tool framework whereby the adopted models represent models of the architecture. Based on the model-based testing approach we propose the MDABT process and the corresponding tool. The tool has been implemented using the Eclipse Epsilon Framework. We illustrate the MDABT tool framework for deriving test cases from different architecture views.

## 1 INTRODUCTION

In general, exhaustive testing is not practical or tractable for most real programs due to the large number of possible inputs and sequences of operations. As a result, selecting set of test cases which can detect possible flaws of the system is the key challenge in software testing. *Model based testing* (MBT) addresses this challenge by automating the generation and execution of test cases using models based on system requirements and behaviour (Utting, et al., 2012). MBT relies on models to automate the generation of the test cases and their execution. A model is usually an abstract, partial presentation of the desired behaviour of a system under test (SUT). MBT can use different representations of the system to generate testing procedures for different aspects of the software systems. Example models include finite state machines (FSMs), Petri Nets, I/O automata, and Markov Chains. A recent trend in MBT is to adopt software architecture models to provide automated support for the test process leading to the notion of model-driven architecture-based testing (MDABT). Software architecture is different from the other design representations since it provides a gross-level representation of the system at the higher abstraction level (Tekinerdogan, 2014). In this paper, we present MD-ArchT, an MDABT tool framework for supporting model based testing using architecture models. The tool takes as input a set of architecture views that can be used to automatically create test cases. In this paper, we focus in particular on using the test cases for checking the architecture-code consistency (Eksi & Tekinerdogan, 2017). We illustrate the MDABT tool framework for deriving test cases from different architecture views.

The remainder of the paper is organized as follows. In section 2, we present the method for architecture driven model based testing that will be implemented using the MD-ArchT tool. Section 3 presents the tool framework. Section 5 the related work and finally section 6 concludes the paper.

## 2 MDABT APPROACH

Models have been widely used in software engineering and support the communication among stakeholders, the analysis, and the guidance in of the overall development process. Model-based testing (MBT) uses detailed models to automatically derive test artefacts (Utting, et al., 2012). There are obvious reasons for adopting MBT including test maintenance, test design and increasing test quality. The MDABT approach that builds on and refines the general MBT process is shown in Figure 1.

Uzun, B. and Tekinerdogan, B.
Model Driven Architecture based Testing Tool based on Architecture Views.
DOI: 10.5220/0006603604040410
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 404-410
ISBN: 978-989-758-283-7

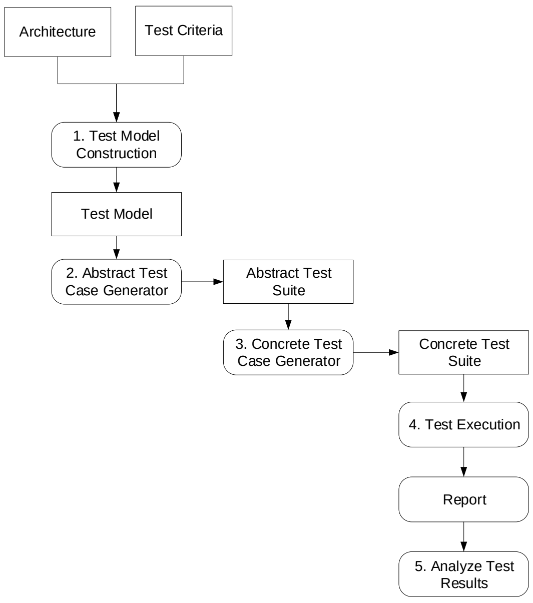
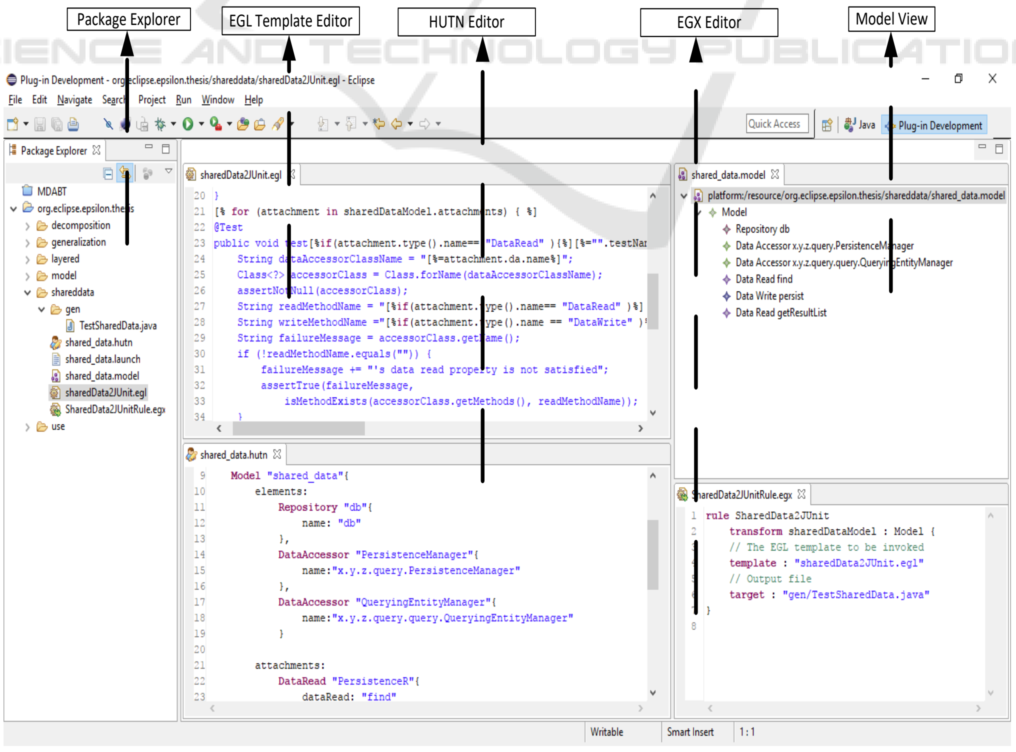


Figure 1: The MDABT Approach.

The foremost issue in MDABT is that the adopted models are not just any design models but architectural models. To derive test artefacts from these architectural models these should be precisely defined. Further it is necessary to properly define the criteria that will be needed to test the system. Testing criteria informs the process about what is to be tested and what is the purpose of the generated tests. Using architecture models along with testing criteria, a test model is created as an intermediate form. Subsequently, an abstract test suite is generated from the testing model which is further used to generate a concrete test suite. The concrete test suite is the actual test set that runs on the system under test. After the test executions, the results are analyzed by a test oracle. Test model construction, abstract and concrete test case generation, test execution and analysis can be automatic or manual depending on the suggested approach.

## 3 TOOL FRAMEWORK

This section provides an implementation of the generic process model presented in Figure 1. The implementation of the tool is based on the Eclipse Epsilon environment that contains languages and tools for code generation, model to model transformation, model validation, comparison, migration and refactoring (Eclipse, 2014). We implemented our tool using Epsilon Generation Language (EGL), Epsilon Generation Runner (EGX), and Human-Usable Textual Notation (HUTN) tool.



```
MDABT
org.eclipse.epsilon.thesis
  decomposition
  generalization
  layered
  model
  shareddata
    gen
      TestSharedData.java
    shared_data.hutn
    shared_data.launch
    shared_data.model
    sharedData2JUnit.egl
    SharedData2JUnitRule.egx
  use
```

```
}
[% for (attachment in sharedDataModel.attachments) { %]
@Test
public void test[%if(attachment.type().name== "DataRead" ){%][%="".testNar
    String dataAccessorClassName = "[%=attachment.da.name%]";
    Class<?> accessorClass = Class.forName(dataAccessorClassName);
    assertNotNull(accessorClass);
    String readMethodName = "[%if(attachment.type().name== "DataRead" )%]
    String writeMethodName ="[%if(attachment.type().name == "DataWrite" )
    String failureMessage = accessorClass.getName();
    if (!readMethodName.equals("")) {
        failureMessage += "'s data read property is not satisfied";
        assertTrue(failureMessage,
            isMethodExists(accessorClass.getMethods(), readMethodName));
    }
```

```
    Model "shared_data"{
        elements:
            Repository "db"{
                name: "db"
            },
            DataAccessor "PersistenceManager"{
                name:"x.y.z.query.PersistenceManager"
            },
            DataAccessor "QueryingEntityManager"{
                name:"x.y.z.query.query.QueryingEntityManager"
            }

        attachments:
            DataRead "PersistenceR"{
                dataRead: "find"
```

```
platform:/resource/org.eclipse.epsilon.thesis/shareddata/shared_data.model
  Model
    Repository db
    Data Accessor x.y.z.query.PersistenceManager
    Data Accessor x.y.z.query.query.QueryingEntityManager
    Data Read find
    Data Write persist
    Data Read getResultList
```

```
rule SharedData2JUnit
    transform sharedDataModel : Model {
    // The EGL template to be invoked
    template : "sharedData2JUnit.egl"
    // Output file
    target : "gen/TestSharedData.java"
}
```

Figure 2: Snapshot of the MDABT Tool.

EGL is a template based model to text transformation language designed in the Epsilon environment. Depending on the provided model code, documents or other textual notations can be derived. EGX is a runner for EGL templates that can parametrize the defined templates in which transformation rule is defined. In the Eclipse Epsilon environment, the creation of models which conforms to predefined metamodels can be modelled using HUTN tool. The execution of EGX file generates a single JUnit test case which can be executed automatically. In Figure 2 a snapshot of the IDE is shown. In the following sub-sections, we describe the process steps in detail.

## 3.1 Architecture View Modelling

Our approach uses architecture views for deriving test cases to check the conformance among the architecture views with the code. Every architecture view conforms to an architecture viewpoint that defines the language for representing the corresponding views. To support model-based testing using architecture views we thus need to define the domain specific languages for the required architecture viewpoints. We did this for each of the architecture viewpoint of the Views and Beyond approach (Demirli & Tekinerdogan, 2011) (Clements, et al., 2010). HUTN is used to generate the view models that are used in the test transformation model construction. The implemented view model in HUTN can be easily converted to view model using the integrated HUTN tool in the Eclipse Epsilon environment.

## 3.2 Defining Test Criteria for Architecture Views

Since we use architecture views for architecture models for testing, we need to define the testing criteria for each of the adopted architecture views. The predefined metamodels for each viewpoint reveals what is to be tested for the respective viewpoint. In our implementation, we have created EGL templates for each architecture view under test. The view criteria are used in the construction of the EGL templates. In addition, the constructed templates are used when generating JUnit test cases. Basically, the transformation rule in the transformation model is applied on the template using the architecture view model.

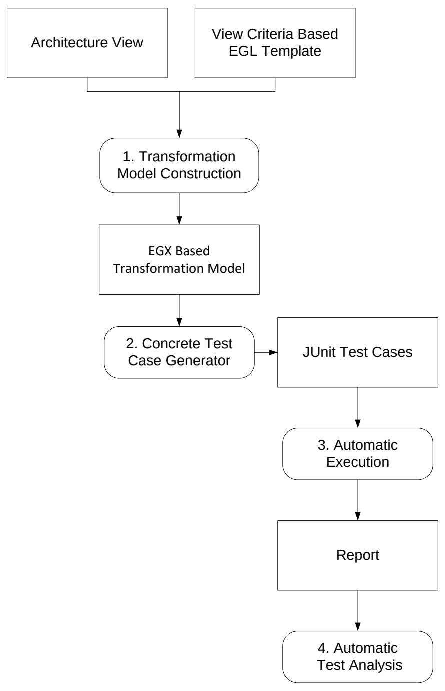

Architecture View
View Criteria Based EGL Template
1. Transformation Model Construction
EGX Based Transformation Model
2. Concrete Test Case Generator
JUnit Test Cases
3. Automatic Execution
Report
4. Automatic Test Analysis

Figure 3: Process Model for Our Approach.

## 3.3 Defining the Transformation Model

Once we have defined the executable models for the architecture views and the required test criteria we developed the generators for automated generation of test cases. For this we adopt the EGX language in which the required transformation rules are defined. The transformation rule details consisting of model to be used, template and its parameters, generated file location and name is defined in the transformation rule. Transformation models are created for each architecture for once for generating the test cases. After the execution of the rule the test cases are generated using given model and template to the given path with its name.

## 3.4 Generating JUnit Test Cases

Test cases for each view are generated consisting of multiple test methods. The generated test cases depend on Java's built-in reflections library. Each generated test case can be imported to the project's path and executed automatically.

## 4 EXAMPLE CASE

In the following we will use the *shared data* and *decomposition* architecture views to derive test test cases for checking the conformance with the code. The shared data viewpoint is used in data-intensive systems in which components interact through a repository. The architecture view specifies the repository, the number and type of data accessors, and data writers. The data or the repository has multiple accessors with different access right as read, write, or read and write.

To develop the domain specific language for this viewpoint we first need to define the metamodel for it. This is shown in Figure 4 (Tekinerdogan & Demirli, 2013).

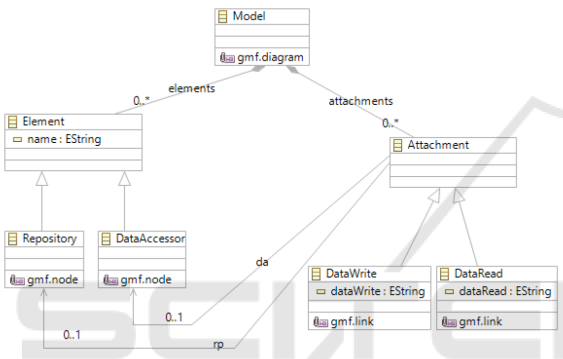
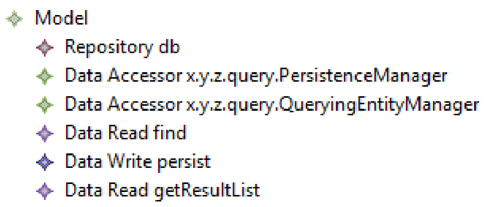


Figure 4: Metamodel for the Shared Data Viewpoint.

Each architecture viewpoint and the defined architecture view enforces some constraints on the system to be implemented. In the shared data viewpoint, we identified the following testing criteria:

- Does each data accessor exist in the code?
- Does each attachment of data accessor exist in the code?

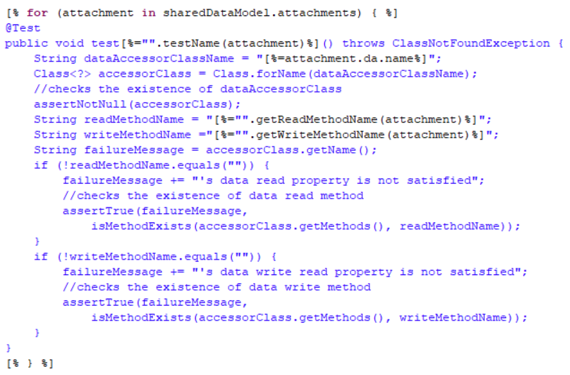
In Figure 5 an example shared data model is shown. In this model, we have two data accessors PersistenceManager and QueryingEntityManager having find, persist and getResultList operations respectively.



Model
- Repository db
- Data Accessor x.y.z.query.PersistenceManager
- Data Accessor x.y.z.query.QueryingEntityManager
- Data Read find
- Data Write persist
- Data Read getResultList

Figure 5 Shared Data Viewpoint Model.

In Figure 6, we have provided our EGL template for the shared data viewpoint that implements the above two constraints as assertions. Considering the model shown above three test methods will be created checking the existence of *persistenceManager* and its operations and *queryingEntityManager* and its operations.



```
[% for (attachment in sharedDataModel.attachments) { %]
@Test
public void test[%="".testName(attachment)%]() throws ClassNotFoundException {
    String dataAccessorClassName = "[%=attachment.da.name%]";
    Class<?> accessorClass = Class.forName(dataAccessorClassName);
    //checks the existence of dataAccessorClass
    assertNotNull(accessorClass);
    String readMethodName = "[%="".getReadMethodName(attachment)%]";
    String writeMethodName ="[%="".getWriteMethodName(attachment)%]";
    String failureMessage = accessorClass.getName();
    if (!readMethodName.equals("")) {
        failureMessage += "'s data read property is not satisfied";
        //checks the existence of data read method
        assertTrue(failureMessage,
            isMethodExists(accessorClass.getMethods(), readMethodName));
    }
    if (!writeMethodName.equals("")) {
        failureMessage += "'s data write read property is not satisfied";
        //checks the existence of data write method
        assertTrue(failureMessage,
            isMethodExists(accessorClass.getMethods(), writeMethodName));
    }
}
[% } %]
```

Figure 6: Shared Data Viewpoint EGL Template Code Snippet for Generation of Test Case Methods.

In Figure 7 the shared data transformation rule is shown. The transformation rule is named *SharedData2JUnit*. It consists of one transformation which takes the model file as input and uses the EGL template (sharedData2Junit.egl) to generate the target output as java file (gen/TestSharedData.java).

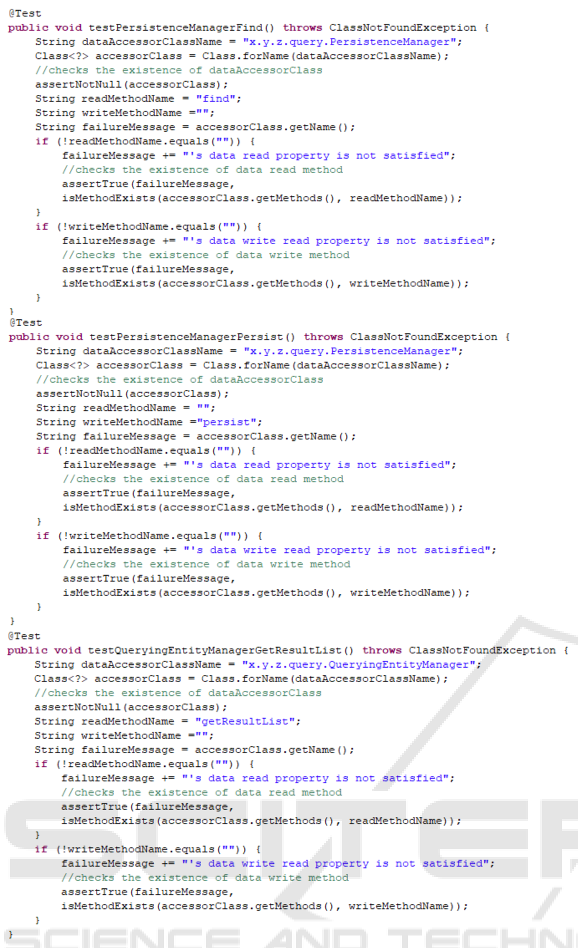
```
rule SharedData2JUnit
    transform sharedDataModel : Model {
    // The EGL template to be invoked
    template : "sharedData2JUnit.egl"
    // Output file
    target : "gen/TestSharedData.java"
  }
```

Figure 7: Transformation Rule for the Shared Data Viewpoint.

The execution of the transformation rule creates a JUnit test case. An example generated code snippet is shown in Figure 8.

The created test case checks it data accessor existence and existence of its data operations. First two test methods check for a data accessor named *persistentManager* and its operations existence. Likewise, third test method checks the existence of *queryingEntityManager* data accessor and its operations.

The decomposition viewpoint deals with concerns of partition of system responsibilities into modules and modules into submodules. It is a containment relation among modules and submodules. In Figure 9 decomposition viewpoint metamodel can be seen (Tekinerdogan and Demirli, 2013).



```
@Test
public void testPersistenceManagerFind() throws ClassNotFoundException {
    String dataAccessorClassName = "x.y.z.query.PersistenceManager";
    Class<?> accessorClass = Class.forName(dataAccessorClassName);
    //checks the existence of dataAccessorClass
    assertNotNull(accessorClass);
    String readMethodName = "find";
    String writeMethodName ="";
    String failureMessage = accessorClass.getName();
    if (!readMethodName.equals("")) {
        failureMessage += "'s data read property is not satisfied";
        //checks the existence of data read method
        assertTrue(failureMessage,
        isMethodExists(accessorClass.getMethods(), readMethodName));
    }
    if (!writeMethodName.equals("")) {
        failureMessage += "'s data write read property is not satisfied";
        //checks the existence of data write method
        assertTrue(failureMessage,
        isMethodExists(accessorClass.getMethods(), writeMethodName));
    }
}
@Test
public void testPersistenceManagerPersist() throws ClassNotFoundException {
    String dataAccessorClassName = "x.y.z.query.PersistenceManager";
    Class<?> accessorClass = Class.forName(dataAccessorClassName);
    //checks the existence of dataAccessorClass
    assertNotNull(accessorClass);
    String readMethodName = "";
    String writeMethodName ="persist";
    String failureMessage = accessorClass.getName();
    if (!readMethodName.equals("")) {
        failureMessage += "'s data read property is not satisfied";
        //checks the existence of data read method
        assertTrue(failureMessage,
        isMethodExists(accessorClass.getMethods(), readMethodName));
    }
    if (!writeMethodName.equals("")) {
        failureMessage += "'s data write read property is not satisfied";
        //checks the existence of data write method
        assertTrue(failureMessage,
        isMethodExists(accessorClass.getMethods(), writeMethodName));
    }
}
@Test
public void testQueryingEntityManagerGetResultList() throws ClassNotFoundException {
    String dataAccessorClassName = "x.y.z.query.QueryingEntityManager";
    Class<?> accessorClass = Class.forName(dataAccessorClassName);
    //checks the existence of dataAccessorClass
    assertNotNull(accessorClass);
    String readMethodName = "getResultList";
    String writeMethodName ="";
    String failureMessage = accessorClass.getName();
    if (!readMethodName.equals("")) {
        failureMessage += "'s data read property is not satisfied";
        //checks the existence of data read method
        assertTrue(failureMessage,
        isMethodExists(accessorClass.getMethods(), readMethodName));
    }
    if (!writeMethodName.equals("")) {
        failureMessage += "'s data write read property is not satisfied";
        //checks the existence of data write method
        assertTrue(failureMessage,
        isMethodExists(accessorClass.getMethods(), writeMethodName));
    }
}
```

Figure 8: Shared Data Viewpoint Generated Test Case Code.

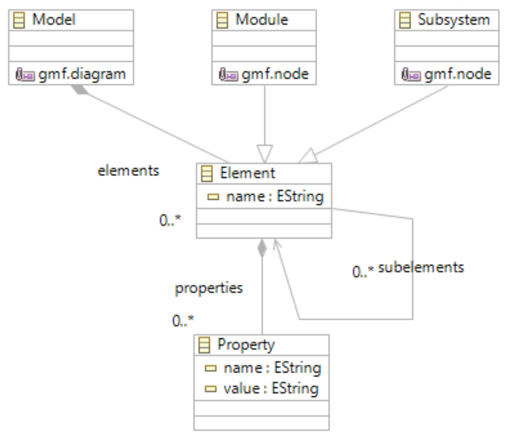
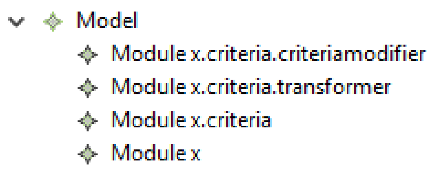


Figure 9: Metamodel for the Decomposition Viewpoint.

In the decomposition viewpoint, we can identify the following testing criteria:

- Does every element in the view model appear in the code?
- Does every subelement in the view model appear in the code?
- Does every subelement exits under corresponding element in the code?

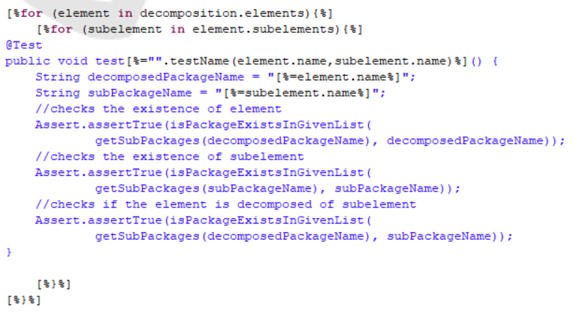
In Figure 10 a sample decomposition model is presented. In this case module *x* is decomposed into *criteria* module which is further decomposed into *criteriamodifier* and *transformer*.



Model
- Module x.criteria.criteriamodifier
- Module x.criteria.transformer
- Module x.criteria
- Module x

Figure 10: Decomposition Model.

Figure 11 shows the EGL template for the decomposition viewpoint. In this viewpoint test case method will iteratively be created for each element's subelements. Three constraints are asserted in the test methods which are: the existence of element, existence of subelement and decomposition relation between element and its subelement. Three test methods will be generated considering the sample model given above. The first test method will check the existence of x, criteria and check the decomposition relation between these elements. Moreover, second and third test methods will both check for criteria's existence and subelements which are criteriamodifier and transformer. Furthermore, second and third test cases will also assert the decomposition relation between these elements.



```
[%for (element in decomposition.elements){%]
    [%for (subelement in element.subelements){%]
@Test
public void test[%="".testName(element.name,subelement.name)%]() {
    String decomposedPackageName = "[%=element.name%]";
    String subPackageName = "[%=subelement.name%]";
    //checks the existence of element
    Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(decomposedPackageName), decomposedPackageName));
    //checks the existence of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(subPackageName), subPackageName));
    //checks if the element is decomposed of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
            getSubPackages(decomposedPackageName), subPackageName));
}

    [%}%]
[%}%]
```

Figure 11: Decomposition Viewpoint EGL Template Code Snippet for Generation of Test Case Methods.

Figure 12 shows the transformation rule for generating test case from decomposition model using the EGL template. The generated file will be named TestDecomposition.java.

The execution of the transformation rule creates again a JUnit test case.

The generated test case has three test methods named testCriteriaDecomposedOfCriteriamodifier, testCriteriaDecomposedOfTransformer and testXDecomposedOfCriteria each method parent and child module existence and parent child relation between modules.

```
rule Decomposition2JUnit
    transform decomposition : Model {
    // The EGL template to be invoked
    template : "Decomposition2JUnit.egl"
    // Output file
    target : "gen/TestDecomposition.java"
}
```

Figure 12: Transformation Rule for Decomposition Viewpoint.

Figure 13 shows the generated test case code snippet for three test methods which are: *testXDecomposedOfCriteria*, *testCriteriaDecomposedOfCriteriamodifier* and *testCriteriaDecomposedOfTransformer*. As shown in the figure there are three assertions are listed in each generated method the first two assertions are checking the existence of *element* and *subelement*, whereas the last assertions check the decomposition relation between *element* and *subelement*. First method checks the assertions for element *x* and subelement *criteria*. Likewise, the second and third test methods checks the assertions for *criteria*, *criteriamodifier* and *transformer*.

```
@Test
public void testXDecomposedOfCriteria() {
    String decomposedPackageName = "x";
    String subPackageName = "x.criteria";
    //checks the existence of element
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(decomposedPackageName), decomposedPackageName));
    //checks the existence of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(subPackageName), subPackageName));
    //checks if the element is decomposed of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(decomposedPackageName), subPackageName));
}
@Test
public void testCriteriaDecomposedOfCriteriamodifier() {
    String decomposedPackageName = "x.criteria";
    String subPackageName = "x.criteria.criteriamodifier";
    //checks the existence of element
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(decomposedPackageName), decomposedPackageName));
    //checks the existence of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(subPackageName), subPackageName));
    //checks if the element is decomposed of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(decomposedPackageName), subPackageName));
}
@Test
public void testCriteriaDecomposedOfTransformer() {
    String decomposedPackageName = "x.criteria";
    String subPackageName = "x.criteria.transformer";
    //checks the existence of element
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(decomposedPackageName), decomposedPackageName));
    //checks the existence of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(subPackageName), subPackageName));
    //checks if the element is decomposed of subelement
    Assert.assertTrue(isPackageExistsInGivenList(
    getSubPackages(decomposedPackageName), subPackageName));
}
```

Figure 13: Decomposition Viewpoint Generated Test Code Snippet *testXDecomposedOfCriteria*, *testCriteriaDecomposedOfCriteriamodifier*, *testCriteriaDecomposedOfTransformer*.

## 5 RELATED WORK

Model-driven architecture based testing (MDABT) has been proposed and discussed in various studies. In Bertolino et al. (2000) an approach is provided for the derivation of test plans from software architecture (SA) specifications. The authors adopt the approach for analyzing the conformance of the architecture with the code. Further they use the SA specification in particular to support integration testing. In subsequent work they adopt sequence diagram as the model for testing the conformance of the architecture and code (Bertolino, et al., 2000), (Muccini, et al., 2004).

In (Jin and Offutt, 2001) formal test criteria are defined based on architectural relations. In Scollo and Zecchini's study (Scollo and Zecchini, 2005), SA based testing is performed on architectural level to test requirements of the system against functionalities of SA. In (Johnsen, et al., 2011), the so-called Architecture Analysis and Design Language (AADL) specifications are used for the verification of software systems. Within the study, both model checking of SA and SA based testing approaches are observed to test code to architecture conformity. In (Keum, et al., 2013), service-oriented applications (SOA) are tested to solve observability and controllability problems that are created by message exchanges between the services that are hidden behind the front service of the system.

In (Lochau, et al., 2014), use SA based testing approaches to variant-rich software systems. Authors addressed the challenge of ensuring correctness of components implementations and interactions with any system variant. The main motivation of the study is to ensure the conformity between code and architecture.

In the study referenced in (Li, et al., 2016), behavioral UML models are used to utilize architecture based testing for generating behavior driven tests for cucumber tool. The proposed approach and tool addresses the concern of conforming given architecture to implementation.

In (Elallaoui, et al., 2016), an automated model driven testing approach which uses UML sequence diagrams is presented.

## 6 CONCLUSIONS

In this paper, we have presented our MDABT approach and the corresponding tool MD-ArchT. We have defined the overall process based on the generic model-based testing process. Different from existing MBT approaches the MDABT process adopts architectural specification as the model to automatically derive test artefacts. The tool has been developed using the Eclipse Epsilon framework based on ecore models. Architecture models represent the architecture views that are represented as specification of the corresponding domain specific language. We have explained the MDABT approach in detail and illustrated the approach and the tool for the shared data viewpoint and the decomposition viewpoint. We have presented the adopted metamodels and the test criteria for these views. Further we have also shown the EGL template and EGX transformation rule. We have illustrated the MDABT approach for testing the conformance between the architecture and the code. Yet, both the process and the tool are in fact generic and can be applied also for different test scenarios. In our future work we will elaborate on this and also integrate behavioral modeling for generating the test artefacts.

## REFERENCES

Antonio, B., Muccini, H., Pelliccione, P. & Pierini, P., 2004. *Model-Checking Plus Testing: From Software Architecture Analysis to Code Testing.* Berlin, Springer.

Bertolino, A., Corradini, F., Inverardi, P. & Muccini, H., 2000. *Deriving test plans from architectural descriptions.* Limerick, ACM.

Clements, P. et al., 2010. *Documenting Software Architectures: Views and Beyond.* 2. ed. s.l.:Addison-Wesley.

Demirli, E. & Tekinerdogan, B., 2011. *Software Language Engineering of Architectural Viewpoints.* s.l., in Proc. of the 5th European Conference on Software Architecture (ECSA 2011), LNCS 6903, pp. 336–343.

Eclipse, 2014. *Epsilon.* [Online] Available at: http://eclipse.org/epsilon [Accessed 1 2 2015].

Eksi, E. & Tekinerdogan, B., 2017. A Systematic Approach for Consistency Checking of Software Architecture Views. *Journal of Science and Engineering,* 19(55.1).

Elallaoui, M., Nafil, K., Touahni, R. & Messoussi, R., 2016. Automated Model Driven Testing Using AndroMDA and UML2 Testing Profile in Scrum Process. *Procedia Computer Science,* Volume 83, pp. 221-228.

IEEE, 1994. *1059-1993 - IEEE Guide for Software Verification and Validation Plans.* [Online] Available at: http://ieeexplore.ieee.org/document/838043 [Accessed 3 1 2014].

IEEE, 2011. *1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems.* [Online] Available at: https://standards.ieee.org/findstds/standard/1471-2000.html [Accessed 3 1 2014].

Jin, Z. & Offutt, J., 2001. *Deriving Tests From Software Architectures.* Hong Kong, IEEE.

Johnsen, A., Pettersson, P. & Lundqvist, K., 2011. *An Architecture-Based Verification Technique for AADL Specifications.* Berlin, Springer.

Keum, C., Kang, S. & Kim, M., 2013. Architecture-based testing of service-oriented applications in distributed systems. *Information and Software Technology,* 55(7), pp. 1212-1223.

Li, N., Escalona, A. & Kamal, T., 2016. *Skyfire: Model-Based Testing With Cucumber.* Chicago, IEEE.

Lochau, M. et al., 2014. Delta-oriented model-based integration testing of large-scale systems. *Journal of Systems and Software,* Volume 91, pp. 63-84.

Mellor, S., Scott, K., Uhl, A. & Weise, D., 2004. *MDA Distilled: Principle of Model Driven Architecture.* 1. ed. s.l.:Addison-Wesley.

Muccini, H., Bertolino, A. & Inverardi, P., 2004. Using software architecture for code testing. *IEEE Transactions on Software Engineering,* 30(3), pp. 160-171.

Muccini, H., Dias, M. & Richardson, D., 2004. *Systematic Testing of Software Architectures in the C2 Style.* Barcelona, Springer, pp. 295-309.

Muccini, H., Dias, M. & Richardson, D., 2006. Software architecture-based regression testin. *Journal of Systems and Software,* 79(10), pp. 1379-1396.

Reza, H. & Lande, S., 2010. *Model Based Testing Using Software Architecture.* Las Vegas, IEEE.

Scollo, G. & Zecchini, S., 2005. Architectural Unit Testing. *Electronic Notes in Theoretical Computer Science,* Volume 111, pp. 27-52.

Tekinerdogan, B., 2014. Software Architecture. In: T. G. a. J. Díaz-Herrera, ed. *Computer Science Handbook, Second Edition, Volume I: Computer Science and Software Engineering.* s.l.:Taylor and Francis.

Tekinerdogan, B. & Demirli, E., 2013. *Evaluation Framework for Software Architecture Viewpoint Languages.* Vancouver, ACM, pp. 89-98.

Utting, M., Pretschner, A. & Legeard, B., 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability,* 22(5), pp. 297-312.

Winbladh, K., Alspaugh, T., Ziv, H. & Richardson, D., 2006. *Architecture-based testing using goals and plans,* New York: ACM.