Testing Model Transformation: Project Report

Amr Al Mallah

School of computer science, McGill University, Montreal Canada.
http://www.mcgill.ca

Abstract. This work describes a proposed framework that is model based for testing model transformations, it also discuss the main challenges, and provide solution to the model comparison challenge. The paper proceeds through the descriptions using a simple example.

Key words: Model transformation, Testing model transformations, testing framework,Model driven testing

1 Introduction

In software engineering, the concept of Unit testing describes a way to validate software systems, in which testers try to partition the system into many specific units. these units are usually coherent and represent certain functionality. Unit testing argues that by testing each of the systems internal pieces as working pieces, then putting these pieces together should yield a working system. In practice unit testing helps reducing integration errors by using its bottom up approach.

In MDE, the idea is to achieve the low-level machine code from the initial high level requirements, by using modeling techniques at the Right level of abstraction, and then define these model transformations as the translation rules between different models until the lowest level transformation which is the code generator. this idea leads us to propose a unit testing framework in the context of MDE, specifically for model transformations. which treats each model transformation step as a coherent unit that could be tests or unit tested.

Traditional unit testing framework in the xunit family (JUnit, NUnit, PyUnit...etc), usually provide extensive utilities for the testers to manipulate the software and the data that they are trying to test. for example it provides methods for set up and clean up before each test case is executed. this is very useful in case of trying to test a feature that depends on the program being in a certain state, or even just test a simple internal step. These frameworks also provide tools to assert and check for many low level data, like different data structures and exceptions. Most if these utilities do not look particularly useful in the case of testing model transformation.

We propose a unit testing framework that would focus on testing model transformations. in such a framework the focus is on testing one function, supposedly an extensive function, which accepts an input model and produces the

transformed model. there is only two data types to deal with, namely the input model's type and the output model's type.

We take the concept of specifying input and output pairs a step further to define criteria matching, which was inspired by the latest behavior testing proposed approaches [Add reference]. and propose to model the testing framework completely to achieve model driven testing of model transformations. with potential usage in a test driven transformation writing approach. we will use a simple case study as a demonstration throughout the work description.

2 Case Study: Traffic to Petri Net transformation

For the sake of demonstration we use a traffic system example, that has road segments with capacity N (will consider N=1 for simplicity), we can connect them using in-ports and out-ports. each road segment can be connected to at most one in-port and one out-port. The traffic system also can contain car generators, from which cars can be generated. and a collector in which cars can be collected, and so leaving the system. for simplicity we will ignore the fork road segments, which are used to branch two roads segments from one, or to join two into one. this formalism as is, allows the modelers to model a traffic system that is static. To add semantics to this system, a transformation into the Petri Nets formalism, is suitable for simulation of the traffic network semantics. and will allow for better analysis of the traffic network.

2.1 The transformation engine

The transformation from the traffic formalism to the petri net one is described using ATOM3 graph grammar rules. each rule defines a LHS and a RHS. when a rule gets executed, it tries to find a match in the input graph (models are represented as graphs), and if it succeeds it replaces that sub graph with the RHS of the rule. each rule can be assigned a priority as to provide certain sequence of execution. since the engine for executing this rules has to be embedded within the modeling environment ATOM3, it hinders automation. a highly modular transformation framework that we are going to use for this example is MoTif. MoTif [3] is a model transformation framework that is based on the DEVS formalism. As to proceed with the example, we built the MoTif model that represent this model transformation traffic to petri nets. The details of this work is irrelevant for this work, as we intend to treat this transformation engine as a black box, and we are only interested in the final output. our intention is test this model to model transformation, which involves different meta models. We use this small example to demonstrate our proposed automated and modular testing framework.

2.2 Model comparison through pattern matching

One of the main challenges in model transformation testing, which most frameworks try to solve, is model comparison. since the framework will need to com-

pare the expected model with the actual model of each test case. Instead of direct matching, we propose a criteria or pattern matching of the models. In other words the tester will have more flexibility to describe the expected result of the input of a test case. for example, consider a test case whose input is a traffic light object and expected output model is simple petri net model with two places linked by two transitions. these places will have an attribute called name, which might not be relevant for the semantics. i.e it doesn't matter if the places names are X and Y, and even the transformation might give these values arbitrary values when it creates those objects. Now consider direct matching. if describe in my expected model that the places names are H and Z, then the test case will fail, even though the models are semantically equivalent. to overcome this we proposed pattern matching instead. so for this example the expected out put will specify that it doesn't care about the names of the places, but rather it's just looking for this model or pattern. in other cases maybe the test case will care about the names in which case the expected model will include them. this is highly powerful, as now we can model and combine different criteria on an expected test case result as we will show. so for example we have several patterns. for example containsTrafficLight criteria, which checks if the model contains a traffic light element that is of the traffic formalism with any name(since it doesn't matter what name it has). and the idea is that building relatively complex criteria and composing them, could allow arguing about the semantics of the model. or in other words achieve semantics comparisons.

2.3 Using TUnit to test

As a proof of concept of this matching criteria proposal, pyunit was used as a base class for a simple testing framework. which contains two assertion methods, shouldHave and shouldNotHave. these methods are introduced as wrappers around the pyunit assert methods, and take care of executing the criteria checks. Five test cases were built and executed, each calling the transformation function on the model to get the actual output model. and then shouldHave and shouldNotHave methods are used to check the criteria's and gets meaningful results, and error messages in the case of unexpected match. for example a test case called testNoTrafficLight executes the transformation in MoTif then calls, shouldNotHave on the output model, and the criteria containsTrafficLight. after all tests get executed the base pyunit testcase class display the results of the tests ran. In the following sections we will describe the modeled version of this testing framework, it provides what's written here and much more. it can scale and describe complex testing scenarios, while staying modular and reusable thanks to the DEVS formalism.

3 TUnit : Modular Timed Model based Transformation Testing Framework using DEVS

The main purpose of any testing framework is to provide utilities for defining control flow structure of test cases executions (sequences, choices, parallelism,

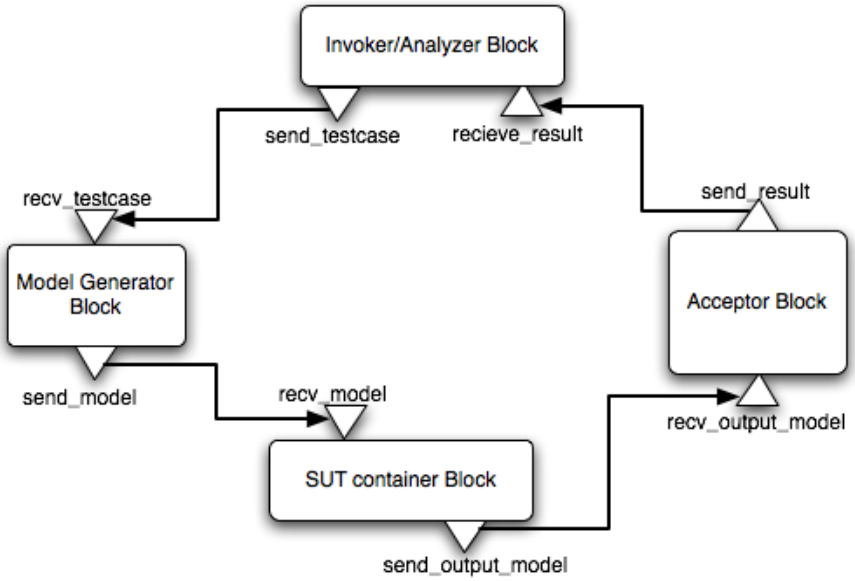

Fig. 1. an overview of the modeled testing framework TUnit

looping) and to associate the expected assertions on the results. In general the execution of test cases is sequential to guarantee isolation. but evidently if the SUT has a context under which it performs parallel execution, then few test cases will need to be executed in parallel simulate the scenario. It is understood that test cases are snap shots of the SUT behavior under certain condition. In other word they are a simulation measurement of the behavior of a system. These factors lead us to propose the DEVS formalism with its clear and exactly defined syntax and semantics to model a testing framework for model transformation testing. to illustrate this we will describe the model of this testing framework (fig 1). We use MoTif's [3] approach of encapsulating models that need to be tested in DEVS events, which will be used to communicate the input model, expected model of each test case and its results. these events could encode other signals to be used in determining different assertions results, which will then on the designated ports to be transmitted between the different DEVS blocks. Atomic DEVS models preform a specific task, make it executing an assertion on a model, or initializing models, as we will see. where as Coupled DEVS models are used to hierarchically construct more complex flow, which is build on composing Atomic DEVS models.

3.1 The Invoker/Analyzer Block

The Invoker/Analyzer block is an Atomic DEVS block, that contains a list of names of the files for the input models of the test cases. It sends the name of the

file to be tested in *testCase* event from its *sendTestcase* out-port. The invoker also receives on its*recvResult* in-port, the result of the test case. this result is encapsulated in an event that's called *finalTestResult*. it contains a Boolean to determine the result and a description message of mismatch errors to help the localization of the errors. it then associate this result information with the last executed test case name in a report. The block keeps sending input names and collecting results until the list of input models has finished.

3.2 The Model Generator Block

The model generator block is an atomic DEVS block which receives on it's in-port *recvTestcase* the *testCase* event. this event contains the file name of the input model that need to be tested. This block is responsible for parsing the input model from the file with the specified name in the input. It then sends this parsed model object encoded in the *testCase* event to the its out port *sendModel*. The model generator block could be customized to parse model files from any format it chooses to support, into an object that is compatible with the SUT. in the case of our example, the SUT is a MoTif model. which understands Atom3's python representation of models. these models in theory could be represented as XML or other format, and then translated into ATOM3 models by the model generator block. the tester could model these input models visually in a modeling environment or textually directly into code.

3.3 SUT Container Block

The SUT (System Under Test) container block, as the name indicates, is an Atomic DEVS block that represent a container for the transformation function/engine that need to be tested. It accepts the on its in put port *recvModel* the event *testCase*. The block then extracts the model object which is encoded in the event, and trigger the transformation function execution on that input model. It then encode the resulting object into the event *testCase* and sends it to its out port called *sendOutputModel*.

3.4 The Acceptor Block

The acceptor block is not an atomic DEVS instance, it's rather a coupled DEVS instance which composite several atomic DEVS building blocks. the actual testing and assertions, is performed in this block. In general the result of the test is determined by the collective result of the results from the different criteria blocks in the from which, it receives the *testCase* event, that encodes the model that need to be tested. (fig 2) it has the following components:

1. Distributor Block : an Atomic DEVS, whose in-port *recvOutputModel* is connected to the in-port of the Acceptor block. its job is to distribute separate copies of the encoded model that needs to be tested into each criteria block, by producing the event *testCase* on its out-port *sendTest*.

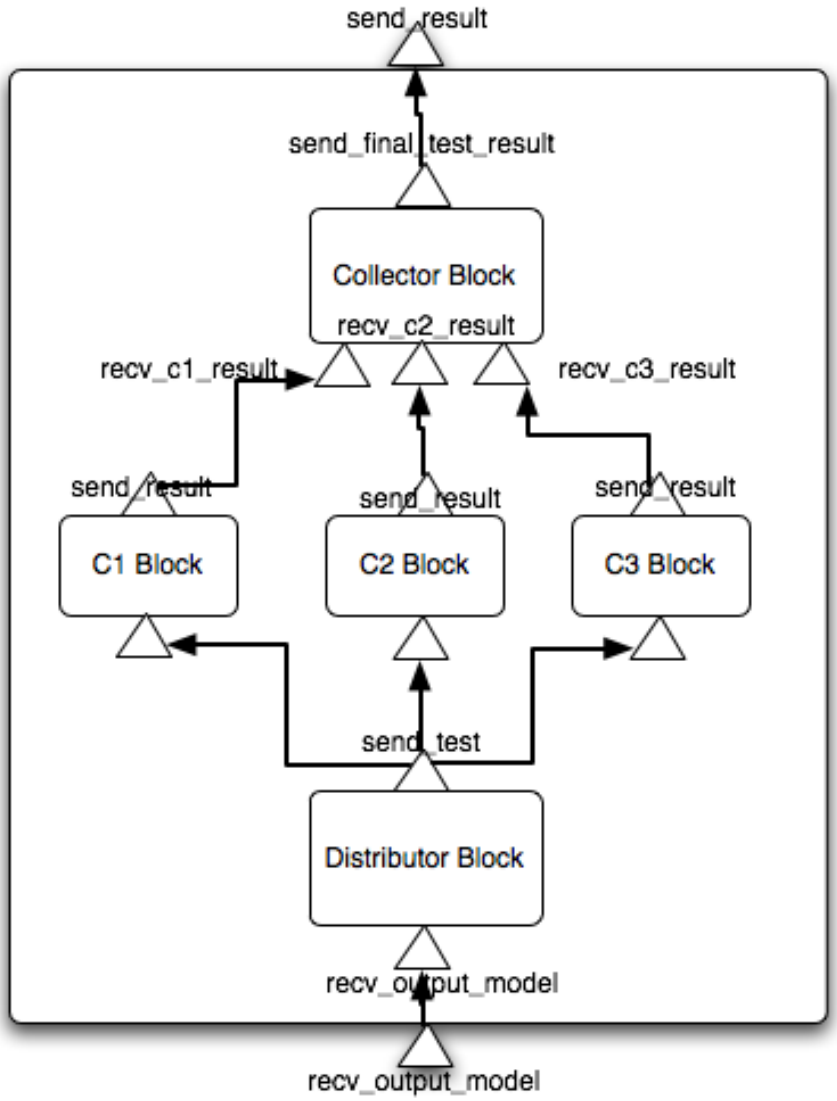

Fig. 2. an overview of Acceptor block in TUnit

2. Criteria Block(s) : each criteria block is an Atomic DEVS, similarly contains an in-port *recvTestModel* to receive the model thats encoded in the event, and execute the criteria check. It also has an out-port *sendCriteriaResult* which it uses to send the result of the criteria check encoded in the event *testCase*. the criteria check can be arbitrary (hand coded, compiled, or even interpreted from some specification on the fly). in the case of our example we allow the testing modeler to model these criteria as models in ATOM3 GG environment. specifically we consider using the LHS of the GG rules to allow testers to model the matching criteria they need to test. this is possible because of a small compiler script that compiles each LHS of each rule into a check method that accepts an Atom3 model as input and returns both the result (about the match) and a description message that helps testers locate the bug source efficiently if any the match was not achieved. An excerpt of the containsNoTrafficLight criteria is shown below

```
class Containstrafficlight:
  def check(self, graph):
    ''' Returns Result, a description string of what went wrong
    '''

    C = {}
    N = {}
    C[1] = 'TR_TrafficLight'
    N[1] = []

    # with the LHS label as key
    for label in C:
      if graph.listNodes.has_key(C[label]):
          N[label] += graph.listNodes[C[label]]


    # Check if all nodes in LHS are in the host graph
    for k in N:
      if not N[k]:
           return False,"Criteria "+self.name+"failed:missing nodes"

    # Verify links between nodes
    M = {} # holds all the matched nodes
    for tr_trafficlight1 in N[1]:
      M[1] = tr_trafficlight1
      break

    # check if all the nodes in LHS are matched
    if len(M) != len(N):
          return False,"Criteria "+self.name+" failed: mismatch"
```

```
return True, "Criteria " +self.name+" was matched successfuly."
```

(The code generated to check a criteria in TUnit, Note that this compiler is based on Eugene Syriani rule compiler, and the that some messages were abbreviated for better formatting on this paper)

3. Collector Block : an Atomic DEVS block, that contains an in-port *recvCriteriaResult* from which it receives the event *testCase* which encodes the result and the name of the criteria it was produced by. The collector block acts as a synchronizer in the sense that it waits for all of the criteria that are attached to it . (its able to figure out that it received all results because it has an in-port for each criteria it's connected to). once all the results have been received from all criteria block, the collector will produce the final result collectively from all the criteria results and subsequently transmitting the final result ,with any description messages, into its out-port *sendFinalTestResult* that is connected to Acceptor block's out-port *sendResult*.

3.5 Extending The model : going beyond

The above described model, as is provide a basic, yet powerful testing functionality. it's describe a testing framework that is completely modeled, as a modular reusable framework. It is however highly extensible. In particular, due to the power of the DEVS formalism, it can be used as building blocks of a much more complex testing scenario. for example we can consider each instance of the complete framework model with all its blocks as coherent units such as a coupled DEVS. Let's call this TestContext block. so now we can build a different TestContext blocks, each will be testing for a specific set of criteria, then control flow can be defined on such blocks, and hence more complex tests scenarios or suites can be constructed. for example, consider the scenario where you want to first test a post condition, and then depending on the outcome of that TestContext you wanna execute a resource intensive TestContext on that is resource intensive, or move to testing other features in a different TestContext. Another possibility is that such a framework can be used to define testing scenarios of different transformations, for example we can extend our Traffic2PN example so that after we execute the TestContext of the resulting PN models, if this step succeeds we forward the results into another TestContext whose transformation will produce reach-ability trees from PN models which will help argue about the semantics.

4 Advantages, Contributions and Conclusion

Our contributions could be summarized in many points. First, the criteria matching described earlier as a basis for determining the overall result of a test case. We have successfully modeled the testing framework itself, which allowed us to go away from low level code, and bring the testing concept to a higher level of

abstraction. This allows for creation of much more complex test scenarios, that would be otherwise tricky to code by hand. and as so helped focusing on the actual testing. The DEVS formalism seems naturally fit for testing, since testing is really a simulation of something, which is what DEVS specializes at.

References

1. Yuehua Lin, Jing Zhang, and Jeff Gray. A Framework for Testing Model Transformations, in Model-driven Software Development, (Sami Beydeda, Matthias Book, and Volker Gruhn, eds.), Springer, ISBN: 3-540-25613-X, 2005, Chapter 10, pp. 219-236, 2005.
2. Benoit Baudry, et al. Model transformation testing challenges. In ECMDA workshop on Integration of Model Driven Development and Model Driven Testing, Bilbao, Spain, July 2006
3. MoTiF : Eugene Syriani and Hans Vangheluwe: Programmed Graph Rewriting with DEVS. In: Manfred Nagl and Andy Schrr (eds), 4th International Conference Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007). Lecture Notes in Computer Science (LNCS). Springer-Verlag, (2008). Kassel, Germany.