UML Associations
Reducing the Gap in Test Coverage between Model and Code

Anders Eriksson[1,2] and Birgitta Lindström[1]
[1]*School of Informatics, University of Skövde, Skövde, Sweden*
[2]*Saab Aeronautics, Linköping, Sweden*

Keywords: Model Coverage, Code Coverage, UML Association, fUML, ALF, MDA, xtUML.

Abstract: This paper addresses the overall problem of estimating the quality of a test suite when testing is performed at a platform-independent level, using executable UML models. The problem is that the test suite is often required to fulfill structural code coverage criteria. In the avionics domain it is usually required that the tests achieve 100% coverage according to logic-based coverage criteria. Such criteria are less effective when applied to executable UML models than when they are applied to code because the action code found in such models contains conditions in navigation and loops that are not explicit and therefore not captured by logic-based coverage criteria. We present two new coverage criteria for executable UML models, and we use an industrial application from the avionics domain to show that these two criteria should be combined with a logic-based criterion when testing the executable UML model. As long as the coverage is less than 100% at the model level, there is no point in running the tests at the code level since all functionality of the model is not yet tested, and this is necessary to achieve 100% coverage at the code level.

1 INTRODUCTION

As software systems become more complex and interconnected, the act of software design and development becomes a great challenge. Model-based development where the software is designed in a modeling language at a platform-independent level and then transformed to an executable implementation is a powerful methodology to help engineers to better understand and develop such complex systems.

With the introduction of executable modeling languages such as xtUML (Mellor and Balcer, 2002) and fUML (OMG, 2013) it has become possible to perform testing at a platform-independent model level. The testing criteria that the software sometimes are required to fulfill might however be much less effective when applied to models than they are when applied to code. Typical examples are the logic-based criteria that are used in the avionics and automotive domain (Eriksson et al., 2012). As a consequence, a developer might overestimate the given coverage for a test suite that is executed on the model and functionality might therefore, remain untested.

The fact that test criteria can be less effective when applied to models than they are if they are applied to code, calls for new criteria that fill in the gap and cover the functionality that traditional logic-based criteria miss when applied to executable platform-independent models. This paper presents two such criteria, which focus on association navigation and iterations in executable UML models.

The paper is organized as follows, Section 2 gives the necessary background to logic-based coverage criteria and UML associations. Section 3 defines the test criteria we propose for executable UML models and discusses their correlation to logic-based criteria, Section 4 describes the industrial case study and its results. Finally, Section 5 presents conclusions and discusses related work, threats to validity and our suggestions for future work.

2 BACKGROUND

This section presents the necessary background and introduces the concepts that we use in this paper. Section 2.1 describes logic-based testing criteria and how these are typically used in a model-driven development environment. Section 2.2 describes UML with a special focus on the associations that are navigated in executable UML models.

Eriksson, A. and Lindström, B.
UML Associations - Reducing the Gap in Test Coverage between Model and Code.
DOI: 10.5220/0005745205890599
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 589-599
ISBN: 978-989-758-168-7

2.1 Software Testing and Logic-based Criteria

There are many test coverage criteria defined for software (Ammann and Offutt, 2008; Zhu et al., 1997). A *test coverage criterion* defines a set of requirements that a test suite must fulfill, i.e., the *test requirements*. For example, to achieve 100% statement coverage, each statement must be executed by at least one of the tests in the test suite. In this case, statement coverage is the test criterion and each statement has its specific test requirement, to execute it.

Logic-based test criteria focus on covering predicate and clauses that are found in software artifacts such as models or source code *Decision coverage* requires that each predicate is evaluated to both true and false at least once by the test suite. *Multiple Condition Decision Coverage* (MCDC) has the additional requirement that each clause in each predicate evaluates to both true and false while the clause is *active* (Chilenski, 2001). A clause is active if changing the truth assignment of the clause changes the truth assignment of the predicate. For example, to achieve MCDC coverage for the predicate (A & B), clause A should be evaluated to true and false while B evaluates to true since clause A is active when B is true. Similarly, B should be evaluated to both true and false while A is true. Some test requirements for MCDC are redundant and for the given example (A & B), MCDC gives three test requirements while decision coverage only gives two.

Logic-based coverage criteria exercise specific evaluations of predicates and clauses but such criteria are not very well suited to test loops. Both MCDC and decision coverage requires that a loop condition is evaluated to both true and false. If the loop condition contains boolean operators, MCDC will require more than two different truth assignments as discussed above. This is however, a poor guarantee for proper testing of a loop since this evaluation can be done by a single test that iterates through the loop a couple of times with different truth assignments to the individual clauses before finally exiting the loop. From a testing perspective, it is reasonable to have at least one test that enters the loop and another test that does not. In addition to this, it is sometimes required that a loop is executed exactly once and more than once. Covering loops is addressed by different graph-based coverage criteria such as *edge-pair*, *round-trip* and *prime-path* coverage (Chow, 1978; Binder, 2000; White and Wiszniewski, 1991; Ammann and Offutt, 2008).

Coverage criteria can be used to generate tests, which is usually what is assumed by researchers. However, in industry the criteria are often used as a metric to evaluate the quality of a given test suite derived from software requirements. Our work is conducted in an industrial setting where development is model driven, following the principles of a *model-driven architecture* (MDA) (Mellor et al., 2004). The design model is automatically transformed by a model compiler to code (e.g., C++) and any further modifications to the software has to be done at the model level. Manual modification of the source code is not allowed. Test cases are created based on software requirements and the coverage criterion is merely used as a metric to estimate whether the test suite fully exercise the structure of the software. In case there are structural elements in the software that the test suite does not cover, then there is some functionality that is still not tested. In our model-driven development context, this can mean one of three things: (*i*) there is a functional requirement for which there is no test case, (*ii*) there is a functional requirement that has not yet been specified, or (*iii*) there is extra (unintended) functionality in the software, which has to be removed. It is often hard to determine which of these three possible situations that is the actual case. Logic-based coverage criteria are often used like this in the aeronautic domain, i.e., as a metric to assess the quality of test suites derived from the software requirements, DO-178C (RTCA, 2011a).

With the use of executable models in model-based development, developers have an excellent opportunity to test and analyze the software in an early stage of development, long before the model is transformed to code. However, unless the model is transformed to code and tested at that level, the developer has little knowledge of what the achieved code coverage is and hence, whether the test suite is good enough. The reason is that the structure of a design model and the resulting code might be quite different. Kirner (Kirner, 2009) identifies some serious transformation issues concerning abstraction and parametrization that could affect the predicates and clauses. The consequence is that the effect of applying a logic-based coverage criterion to the design model is likely to be less effective than when the same criterion is applied to the code since the code might contain a different set of predicates and clauses to be covered than the model. Moreover, if there are predicates and clauses in the code, which are not covered by the tests, then there is functionality that has not been tested when executing the tests at the platform-independent design level.

Having to transform the model to code in order to estimate the quality of a test suite is problematic for several reasons. One of the reasons is that the transformation to code might be time consuming but the main reason is that the developer should ideally be allowed to perform analysis at the same level he or she develops the software and the test suite, since iterating between abstraction levels can be hard. We address the overall problem of measuring the quality of a test suite, which is required to fulfill a logic-based code coverage criterion, without having to first transform the model to code.

2.2 UML and Associations

There are empirical evidence available showing that class diagram, of all the diagram types provided by UML, is the most widely used in practice by the industry which uses UML (Hutchinson et al., 2011). A class diagram is a static diagram that captures and abstracts the *structure* within a problem domain, and consists of several classes connected with relationships. In UML, there are different types of relationships such as associations, generalizations and dependencies. In this study we focus on the association, which is one of the most powerful constructions in UML and is often misunderstood (Selic, 2012). Moreover, it is the association relationship that has shown to be a source to the differences between the sets of predicates and clauses in model and code (Eriksson et al., 2012).

To capture the dynamic *behavior* within a problem domain, state machine diagrams and activity diagrams are widely used (Hutchinson et al., 2011). In contrast to the structural model, which specifies constraints for allowable configuration of instances, the behavioral model specifies how the state of instances changes over time. A state-machine can be used for specifying the overall life-cycle of an active class, where each state is associated with an activity which coordinates the execution of actions. An instance of a class is called an *object*, and an instance of an association is called a *link*.

Many model-based testing techniques do not deal with actions in behavioral models (Planas et al., 2009). These actions represent the primitives used in behavior models, which in addition can be used for building high-level constructs in an action language (OMG, 2011; Mellor and Balcer, 2002). The main effect of including actions in behavior models is that instances, e.g., objects and links, are *created* and *destroyed* during run-time, which makes the overall system state change continuously over time. Previous experimental studies (Eriksson et al., 2012; Eriksson et al., 2013) have highlighted that there are implicit predicates in association navigation, sequences of *ReadLink* actions expressed in an action language and that these implicit predicates are not guaranteed to be covered by any logic-based coverage criteria. The implicit predicates are derived from the constrained rules specified on the class diagram. This result shows that current logic-based coverage criteria are insufficient for testing the behavior of the platform-independent and executable models. The result can also be interpreted as there is little benefit in measuring coverage at the code level before all the implicit predicates as well as all the explicit predicates are covered in the behavioral model at a platform-independent level, since less than 100% coverage of these at the model level means that there will be less than 100% coverage at the code level.

The characterization of an association in UML is qualified by several elements:

- ***Identity:*** This makes the association uniquely identifiable within the class diagram.
- ***Multiplicity:*** This defines the number of objects of the participating classes at some point in time. Such that one object at one end connects with *"one"* or *"many"* object(s) at the other end. Denoted as [1..1] and [1..*] at each end of the association in the class diagram.
- ***Conditionality:*** This defines if the participation is conditional, i.e., at some point in time there might not be any participating objects. Such that one object at one end connects with *"zero or one"* or *"zero or many"* object(s) at the other end. Denoted as [0..1] and [0..*] at each end of the association in the class diagram. We refer to this as different *configurations*.
- ***Role:*** This describes how the objects logically participates in the association, and there is a role description for each end of the association.

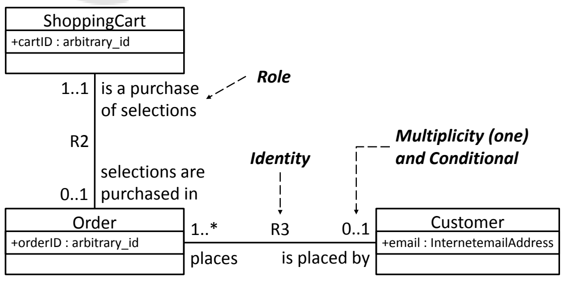


Figure 1: Part of the Online Bookstore Domain class diagram,(Mellor and Balcer, 2002).

In the UML standard (OMG, 2011), all the elements for an UML association is completely described. In our work, associations are restricted to binary associations, and generalizations which are noted as disjoint and complete as described in (Mellor and Balcer, 2002). In Figure 1, is an excerpt of a class diagram given by Mellor and Balcer (Mellor and Balcer, 2002) shown with the qualified elements for an association pointed by the dashed arrows in the figure.

3 TEST CRITERIA

This Section presents two coverage criteria that are meant to be used in combination with a logic-based coverage criteria as a metric for test coverage when testing executable UML models. The two new criteria together cover the implicit predicates that logic-based criteria miss. Section 3.1 defines a criterion that targets navigation and discusses its relation to logic-based criteria. Section 3.2 defines a criterion that targets iterations and discusses the relation between our two proposed criteria.

3.1 All Navigation Criterion

A *navigation step* is a directed navigation of an association identified by its identity. A navigation step starts at one end of an association with the navigation direction towards the other end specified by its role description. This follows the *look-across* semantics of how to read relationships, to which UML adheres (Chen, 1976). A navigation step s is *conditional* if the association navigated is conditional in the direction of the navigation, and we say that the predicate *Empty(s)* evaluates to true iff the collection of participating objects at the end of s is empty.

An *association path* is a contiguous sequence of one or more navigation steps, which specifies the path from a start class to a destination class.

A *selection query* is an expression, which when evaluated read the *links* for an association path in order to identify the current set of objects that are related to an object or set of objects at some point in time.

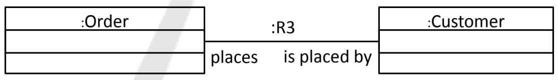
Let Q be the set of selection queries. For each selection query $q \in Q$, let A_q be the association path traversed in response to q and let C_q be the set of conditional navigation steps in A_q. Finally let $c_i \in C_q$ be the i^{th} conditional navigation step in A_q. A conditional navigation step c_i *determines* q iff the set of objects returned upon evaluation of q differs for $Empty(c_i)$ and $\neg Empty(c_i)$.

Definition 1 All Navigation (ANAV): *For each query $q \in Q$ and each conditional navigation step $c_i \in C_q$, choose all navigation steps $c_j \in C_q$, where $j \neq i$ so that c_i determines q. For each such configuration, ANAV has two test requirements, $Empty(c_i)$ and $\neg Empty(c_i)$.*

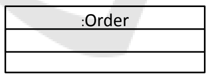
3.1.1 Relation to Logic-based Criteria

This section focuses on the correlation between our proposed criterion ANAV and the logic-based criteria. We show a set of motivating examples, which illustrate the need to combine logic-based criteria with ANAV and discuss the overlap between them. The overlap is not straight-forward since the test requirements come from different sources. Logic-based test criteria gather the test requirements from predicate and clauses found in conditional statements, e.g., in if-statements, while ANAV gather the test requirements from selection queries. Still, when it comes to the tested configurations, we can see that there is an overlap in what the criteria guarantee. A select query, including conditional navigation step(s) in an action language such as OAL [1] or ALF (OMG, 2013) is typically followed by an if-statement, which checks that the resulting object or set of objects is not empty before further processing. Consider the following action code in OAL:

```
1 select one Customer related by
    Order->Customer[R3];
2 if (not_empty Customer)
3    <Statements>
4 end if;
```



(a) $\neg Empty(R3)$



(b) $Empty(R3)$

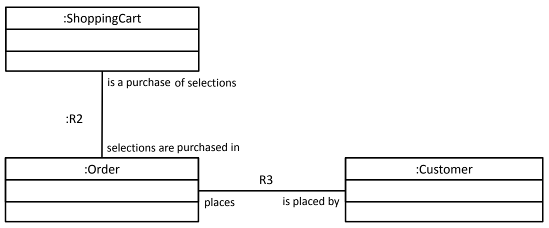
Figure 2: Valid configurations of instances in some point in time.

There are two possible configurations for this piece of action code. $\neg Empty(R3)$ and *Empty(R3)* as shown in Figures 2a and 2b. ANAV gives two test requirements based on the instruction on line 1, while a logic-based criteria gives two test requirements based on the predicate on line 2. The predicate on line 2 has a single clause and hence, will only give two test requirements (true and false) independent on which logic-based criteria we apply. The above action code is an example where the overlap between the two criteria ANAV and Predicate is 100%. Independent of whether we apply ANAV on line 1 or a

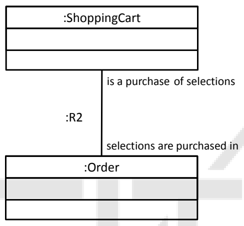
[1]http://www.xtuml.org

logic-based criterion on line 2, both configurations in Figures 2a and 2b will be covered. The same goes for all configurations where there is only one conditional navigation step. Let us consider action code for a configuration with more than one conditional step:

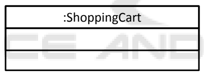
```
select one Customer related by
  Cart->Order[R2]->Customer[R3];
if (not_empty Customer)
   <Statements>
end if;
```



(a) ¬Empty(R2)∧¬Empty(R3)



(b) ¬Empty(R2)∧Empty(R3)



(c) Empty(R2)∧Empty(R3)

Figure 3: Valid configurations of instances in some point in time.

A logic-based test criterion will still give two test requirements for this code. However, these two test requirements will in this case, not be sufficient to cover all configurations. There are three possible configurations for this code as shown in Figures 3(a), 3(b) and 3(c). The logic-based criteria give a guarantee to cover configuration 3(c) and one of the others but not all three. In contrast, ANAV will give four test requirements (of which one is redundant) and guarantees coverage of all three configurations. Hence, this is an example that illustrates that logic-based test criteria do not subsume ANAV.

Let us finally rule out the possibility that ANAV subsumes any of the logic-based criteria. This is simply not possible due to the fact that ANAV focuses on the navigation via the association paths only and there can be other constructs in the action code that contain predicates. In our two illustrating examples, the if-statements both check that the object set returned by the queries are not empty. There can of course be other conditional statements in an action code. In such cases, a logic-based criterion will give a set of test requirements while ANAV will not.

The above examples show that ANAV and logic-based criteria should be combined in order to get sufficient coverage of the potential configurations when testing the platform-independent behavioral model.

3.2 Iteration Criterion

Loops should be executed at least by a test that enters the loop and by a test that does not enter the loop, in order to be properly tested. Neither the proposed ANAV nor the logic-based coverage criteria guarantee sufficient coverage for all loops that can be found in executable models. A single test can fulfill the logic-based criteria for a predicate in a loop condition by iterating through the loop fulfilling one test requirement for each iteration. Hence, logic-based coverage criteria do not guarantee that there will be a test that reaches the loop and does not enter it. Moreover, there are loops in the action code where the loop condition is implicit, i.e., there is no predicate to be covered by logic-based criteria. Such loops iterates over a set of objects that are returned by a query but the stopping criterion is not explicitly stated as a predicate.

Hence, a logic-based coverage criterion would not give any test requirement at all for such loop. Such implicit conditions will become explicit as predicates when the model is transformed to source code. Consider the following action code in OAL:

```
select many Orders related by Carts->Order[R2];
for each Order in Orders
   <Statements>
end for;
```

The above set-iteration statement has no explicit stopping criteria for the loop. Hence, a logic-based criterion would not guarantee that this loop is covered by any test.

Definition 2 All Iteration (ITER): *For each object set-iteration statement, ITER has two test requirements: The test suite must contain at least one test that executes the loop zero times and one test that executes the loop at least one time.*

3.2.1 Relation to ANAV

When it comes to loops in action code, there is an overlap in terms of what coverage ANAV and ITER guarantee. Consider this action code again:

```
1 select many Orders related by Carts->Order[R2];
2 for each Order in Orders
3     <Statements>
4 end for;
```

The navigation step traversing R2 is conditional [0..1] and therefore, ANAV applies to line 1 and ITER applies to line 2 but ANAV will ensure that there is one test returning an empty set of objects and another test returning a set that contains at least one object for this query. In the first case, the test will not enter the loop and in the second case the loop will be executed at least once. Hence, although ANAV focuses on covering navigation and ITER focuses on covering the implicit loop conditions, we do have an overlap in terms of loop coverage. Let us consider another example.

```
1 select many Carts related by
    Orders->ShoppingCart[R2];
2 for each Cart in Carts
3     <Statements>
4 end for;
```

In this case, the navigation step traversing R2 is unconditional in the direction of navigation [1..1] and therefore, the set of conditional navigation steps C_q would be empty. Hence, ANAV would not generate any test requirement for the query on line 1 and it is therefore, only ITER that gives any guarantee for loop coverage. For this example, ITER will only give one test requirement, to execute the loop at least once. The second test requirement (to execute the loop zero times) will be discarded as *infeasible* since there will be at least one element in the set of objects returned by the query. Let us consider a last example.

```
1 select many Carts from instances of ShoppingCart;
2 for each Cart in Carts
3     <Statements>
4 end for;
```

ANAV is not applicable in this case, because there is no navigation. The selection is done directly from the extent of objects for the class ShoppingCart. As in the previous example, it is only ITER that gives any guarantee for loop coverage. ITER will give two test requirements, one to not enter the loop at line 2 (ensuring there is a test returning an empty set of objects) and another to enter the loop (ensuring there is a test returning a set that contains at least one object for this selection).

4 EVALUATION

The work presented in this paper is conducted in an industrial context where we have used six applications from the avionics domain. Several of these are from an updated version of the Gripen fighter [2]. Table 1 shows a summary of the six applications, which all are modeled in xtUML. The number of classes specified in the class diagrams for each application is represented by column Classes in Table 1. Columns States and Operations show the number of places in the applications where their behavior is specified in the object action language (OAL[1]). The six applications have been used in a previous study to investigate the transformation impact on the number of test requirements for logic-based criteria.

Table 1: Summary of experimental subjects.

Applications	Classes	Action Source	
		States	Operations
A1	61	117	193
A2	66	117	216
A3	12	15	46
A4	18	15	34
A5	36	81	140
A6	444	2	633
Total	637	347	1262

We know from a previous study (Eriksson et al., 2013) that the number of test requirements for decision (or predicate) coverage of our applications increases by 69% as an average when the xtUML model is transformed to C++ code. The increase shows that these applications contain a large number of implicit predicates that become explicit during transformation. However, we have also shown that if the implicit predicates residing in the conditional links in navigation of associations, and in the for-each loops are covered by test requirements at the model level, the increase in number of test requirements after transformation to code drops to zero for four of the six applications and less than 0.5% as an average.

The purpose with the new model coverage criteria *all-navigation* (ANAV) and *all-iteration* (ITER) is to cover the implicit predicates in UML behavioral models that becomes explicit as the model is transformed to code. The new criteria target these implicit predicates at the model level. ANAV ensures that all conditional links in an association path are evaluated to both *not_empty* and if possible *empty* (not including any infeasible navigation steps) in the direction of navigation. This will guarantee that we have tested

[2]http://saab.com/air/gripen-fighter-system/gripen/gripen/

at least two possible configurations of instances satisfying each conditional link in the association path during run-time. ITER ensures that for-each loops are executed both zero times and and least once and will guarantee that we have tested the two possible evaluations of all implicit conditions that reside in for-each loops.

Hence, our proposed criteria cover most of the implicit predicates in our six applications and by combining them with the logic-based criterion decision (or predicate) coverage, which covers the explicit predicates, we will as an average capture 99.5% of the test requirements for code-level decision coverage of our six applications at the model level. Hence, functional testing by simulation can be performed at the same level of abstraction as the design is conducted while the quality of the test suite can be estimated without the need of first transforming the model to code.

Moreover, if 100% coverage of a logic-based criterion must be achieved at code-level there is no point in actually executing tests at the code level until the functional testing at model level has achieved 100% model coverage with respect to the logic-based criterion as well as both ANAV and ITER. The reason is that as long as there are implicit predicates that remains to be covered in the model, there will also be explicit predicates in the code that the tests will not cover.

Table 2: Test requirements for application A3.

	Number of Test Requirements			
	Model level			**Code level**
Action Source	ANAV	Predicate	ITER	Predicate
if()		146		146
elif()		14		14
while()		6		6
set-iteration()			44	44
select-1-step()	86			
select-2-step()	32			32
select-3-step()	24			24
select-where()		48		48
select-others()				10
Total number	142	214	44	324

Our previous studies have focused on the transformation impact on the number of test requirements with respect to logic-based coverage criteria. In order to evaluate the effect of using our proposed criteria ANAV and ITER in combination with a logic-based criteria, we take a closer look at one of the applications, A3. Table 2 shows the details of the set of test requirements for application A3. We can see the distribution of test requirement for ANAV, Predicate and ITER respectively in column Model level. Column Code level shows the distribution of test requirements for Predicate at code level. Each row in Table 2 corresponds to a type of action source in the model. For example, we can see that 44 of the test requirements that need to be covered at code level in order to get Predicate coverage come from implicit predicates in set-iteration statements in the action code. We can also see in the same row that these 44 test requirements are only captured by ITER at the model level. The action sources named select-x() represents different kind of navigation statements; select-1-step, select-2-step and select-3-step represents selection queries including one, two or three conditional navigation steps. The select-where represent predicates which is evaluated to further limit the number of objects returned by a selection query. Finally, the select-others represent selection queries that are unconditional, and includes navigation steps starting with set of objects, which enable generation of for-loop statements including exit criterion as a predicate in the C++ code.

It is clear from Table 2 that all types of sources (i.e., all rows) contributes to the test requirements for Predicate coverage of the C++ code, except for the select-1-step where the explicit predicate is only generated if further navigation steps are followed after a conditional navigation step. However, in this case is the modeler responsible for explicitly check the selection result before further usage. Hence, these test requirements should be tested at the model level.

It is also clear that all three criteria ANAV, Predicate and ITER have to be applied to capture all the test requirements at the model level. Finally, we can see that the total number of test requirements that we get for the model is higher than for the code. The reason for this is that there is sometimes an overlap between the three model criteria as we have described in Section 3.

In order to study the difference in model and code coverage, we ran a test suite for application A3 and measured the coverage at both model and code level when the ANAV and ITER criteria were applied. The tools used in this experiment are the BridgePoint[1] for the UML modeling environment and the model-compiler, and for measuring structural code coverage the tool Cantata++[3] was used.

To our knowledge, there is no available tool, including BridgePoint, that enables measuring of model coverage during simulation of behavioral UML models using actions. Therefore we designed and implemented a plugin to the model compiler to be used in our test framework. The plugin enables measuring the model coverage during simulation. Both ANAV and ITER are included as metrics. Moreover, logic-based and model-based coverage criteria according to DO-

[3]http://www.qa-systems.com/cantata.html

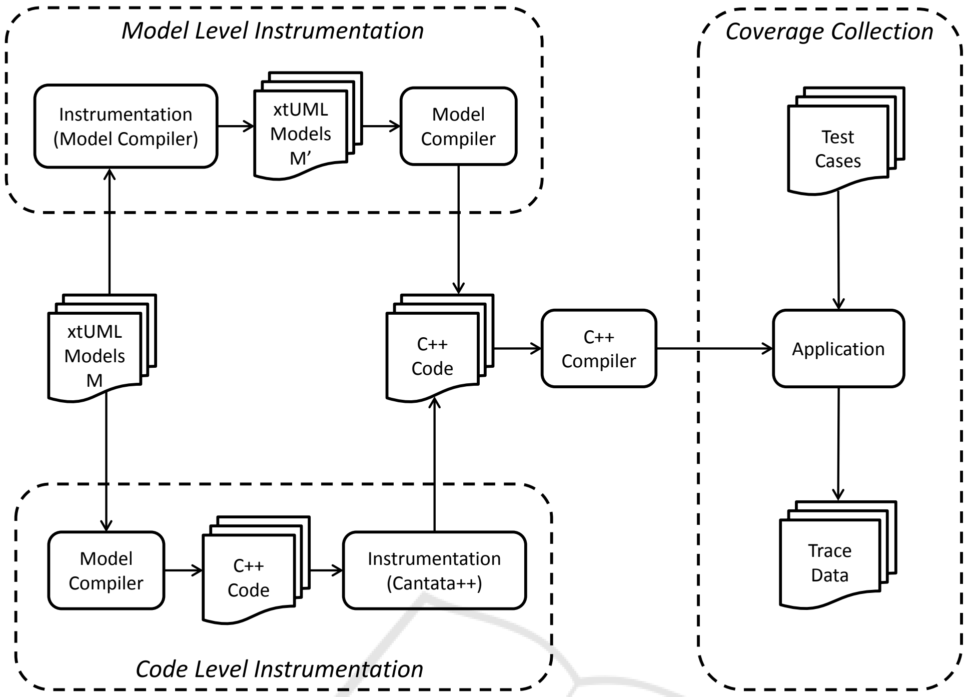


Figure 4: The setup for measuring test coverage.

178C (RTCA, 2011a) and DO-331 (RTCA, 2011b) are also included. The design of the framework is based on our previous work (Eriksson et al., 2013) and work by Kelly et al. (Hayhurst and Veerhusen, 2001). The idea is to transform a design model *M* to an instrumented model *M'* before generating C++ code automatically via the model-compiler, as shown in Figure 4. Note, the simulation of the model behavior is performed through a compiled version instead of using the model interpreter built into BridgePoint, this is only for practical reasons and will not influence the outcome.

Table 3: Test coverage at model level and code level for application A3.

	Number of Covered Test Requirements			
	Model level			Code level
	ANAV	Predicate	Iteration	Predicate
Total	142	214	44	324
Executed	96	144	28	237
Coverage (%)	68	67	64	73

Table 3 shows a snapshot of the coverage achieved by the current version of the test suite for application A3. The application is under development and the test suite is therefore, not yet complete. Hence, the coverage is less than 100% for both code and model. The coverage at the model level (67% as a total) is slightly less than the coverage level for the code (73%). This has to do with the overlap between the criteria for model coverage that we have seen and discussed in Section 3. Since two overlapping test requirements are covered by exactly the same test cases, the difference in coverage between model and code is reduced as new test cases are executed and the coverage is increased. Hence, the closer we get to the goal of 100% coverage, the higher is our confidence that the coverage measured for the model corresponds to the coverage we will get for the code.

Adding the criterion *all-navigation* to the model level becomes even more important in problem domains where conditional associations are used for expressing complex application logic, which constraints the behavior during run-time. An increase in the number of test requirements derived from the *all-navigation* and *all-iteration*, can also be interpreted as a measure of the degree of *application logic complexity*. As an example, the part of the in-house model-compiler that transforms the OAL constructs, and which we have used in our studies is modeled in xtUML and transformed into C++. The traditional abstract syntax tree that represents a programming language is here represented in xtUML as a meta-model, and the model-compiler behavior is specified in OAL code. Table 1 shows a summary of the model-compiler, application A6. A significant amount of structural constructs, and statements in the object action language is conditional or optionally, and these rules are formalized as conditional associations between classes in the meta-model for OAL. Which in this case with large likelihood is that the model-compiler have a higher application logic complexity than the application A3, Table 1, and therefore needs more configurations, combination of instances, to cover the specified functionality.

5 CONCLUSIONS AND DISCUSSION

We address the overall problem of estimating the quality of a test suite that is required to fulfill code coverage according to logic-based coverage criteria, in a model-driven development environment where both design and test activities are performed using models of the software. We have presented two new coverage criteria for executable UML models ANAV and ITER, and shown that these two should be combined with the logic-based criterion when testing the executable model. As long as the coverage is less than 100% at the model level, there is no point in running the tests at the code level since all functionality of the model is not yet tested, and this is necessary to achieve 100% coverage at the code level.

5.1 Threats to Validity

When it comes to validity threats of case studies, it can usually be argued that the study might be too small and the software not representative. However, the coverage criteria that we suggest applies to executable UML models and the models we use cover all types of elements (e.g., associations and multiplicity) and structural constructions (e.g., loops and if-statements) that are allowed in the language for executable UML, i.e., xtUML and fUML. Moreover, the models are real applications developed by different teams in the aeronautics domain, which shows that the constructs that our criteria target, is used by different developers. Finally, one of our applications is an example of a very complex piece of software. A6 is a model compiler that takes an xtUML model as input and generates the C++ code. The model elements and constructs that our proposed criteria target are even more frequent in A6 than in any of the other models.

A potential threat to validity might be the fact that the same model compiler is used for all applications. A model compiler is free to translate the model as long as the semantics and functionality of the software is maintained. Hence, another compiler can give a slightly different translation to code than the one we used. However, our models conforms to and cover the language of executable UML models and the elements and structures that ANAV and ITER target are present in such models. Independent of model compiler, the implicit predicates that we find in these elements and structures will be translated to explicit predicates. The number of predicates and their locations in the code might however, differ between compilers.

5.2 Related Work

There are several papers on how to implement UML associations in programming languages (Akehurst et al., 2007; Goldberg and Wiener, 2011; Diskin et al., 2008), and also authors that argue that relations as a semantic constructs should be an already built-in construct in object-oriented languages (Rumbaugh, 1987; Nelson et al., 2008). The semantics of associations is also investigated thoroughly in (Milicev, 2007). All this work emphasize the needs and importance of a correct implementation and interpretation of relationships, which is a strong argument for the need of our new *all-navigation* criterion.

The new criterion *all-navigation* can be seen as a further development of the *association-end multiplicity (AEM)* criterion and the *generalization (GN)* criterion defined by (Andrews et al., 2003). When testing a model, the *AEM* criterion will ensure that the model is instantiated so that both boundary and non-boundary occurrences of association links between runtime-objects are created. While the *GN* criterion will require the model to be tested with instantiations of each type of the specialization. The difference between the criteria by (Andrews et al., 2003) and the criterion *all-navigation* is where in the model the model elements are covered. The criteria *AEM* and *GN* is fulfilled globally among all statements in the behavioral specification, while the *all-navigation* criterion have to be fulfilled locally within the same association navigation statement resulting in an thoroughly testing of that particular statement.

The idea that loops should be covered thoroughly is not new, it was recognized already back in the 70's (Howden, 1978) and several graph-based criteria such as *edge-pair*, *round-trip* and *prime-path* coverage have been defined to ensure that loops are thoroughly covered by tests (Chow, 1978; Binder, 2000; White and Wiszniewski, 1991; Ammann and Offutt, 2008). Our proposed criterion *all-iteration* is needed together with *all-navigation* and the logic-based criteria to fully support model coverage of executable UML models regardless of the implementation of actions used for association navigation or iterations over set.

5.3 Future Work

The plan is to extend our method to include traceability functionality of model constructs to code constructs needed for coverage analysis. The traceability functionality should be platform independent and make it possible to compare test requirements derived from the model level and the code level. An evaluation of the usability of the complete method should also be conducted.

In the long-term, the aim is to support model coverage analysis according to DO-331 (RTCA, 2011b), and take credit for parts of the verification activities already at model level by simulation instead of at the code level.

ACKNOWLEDGEMENTS

This research was partially supported by Research Grant 2013-01215 from the VINNOVA (Swedish Governmental Agency for Innovation Systems).

We thank Martin Nilsson at Saab Aeronautics for his collaboration with the implementation of the xtUML model coverage tool used in this research.

REFERENCES

Akehurst, D., Howells, G., and McDonald-Maier, K. (2007). Implementing associations: Uml 2.0 to java 5. *Software & Systems Modeling*, 6(1):3–35.

Ammann, P. and Offutt, J. (2008). *Introduction to software testing*. New York: Cambridge University Press, ISBN 978-0-521-88038-1.

Andrews, A., France, R., Ghosh, S., and Craig, G. (2003). Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13(2):95–127.

Binder, R. (2000). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

Chen, P. P.-S. (1976). The entity-relationship modeltoward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36.

Chilenski, J. J. (2001). An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report, Office of Aviation Research.

Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187.

Diskin, Z., Easterbrook, S. M., and Dingel, J. (2008). Engineering Associations: From Models to Code and Back through Semantics. In *TOOLS (46)*, pages 336–355. Springer.

Eriksson, A., Lindström, B., Andler, S. F., and Offutt, J. (2012). Model Transformation Impact on Test Artifacts: An Empirical Study. In *Proceedings of the 9th workshop on Model-Driven Engineering, Verification and Validation*.

Eriksson, A., Lindström, B., and Offutt, J. (2013). Transformation rules for platform independent testing: An empirical study. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 202–211.

Goldberg, M. and Wiener, G. (2011). Generating Code for Associations Supporting Operations on Multiple Instances. In *Evaluation of Novel Approaches to Software Engineering*, pages 163–177. Springer.

Hayhurst, K. J. and Veerhusen, D. S. (2001). A practical approach to modified condition/decision coverage. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1B2–1. IEEE.

Howden, W. E. (1978). An evaluation of the effectiveness of symbolic testing. *Software: Practice and Experience*, 8(4):381–397.

Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. (2011). Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM.

Kirner, R. (2009). Towards preserving model coverage and structural code coverage. *EURASIP Journal on Embedded Systems*, 2009:6:1–6:16.

Mellor, S. J. and Balcer, M. J. (2002). *Executable UML: A Foundation for Model Driven Architecture*. Boston: Addison Wesley, ISBN 0-201-74804-5.

Mellor, S. J., Scott, K., Uhl, A., and Weise, D. (2004). *MDA Distilled: Priciples of Model-Driven Architecture*. Boston: Addison Wesley, ISBN 0-201-78891-8.

Milicev, D. (2007). On the semantics of associations and association ends in uml. *Software Engineering, IEEE Transactions on*, 33(4):238–251.

Nelson, S., Noble, J., and Pearce, D. J. (2008). Implementing first-class relationships in Java. *Proceedings of RAOOL*, 8.

OMG (2011). Unified Modeling Language (UML), Superstructure, version 2.4.1. http://www.omg.org/spec/UML/2.4.1.

OMG (2013). Action Lanaguage for Foundational UML (ALF), version 1.0.1. retrived September 14, 2011.

OMG (2013). Foundational Subset of Executable UML (FUML), version 1.1.

Planas, E., Cabot, J., and Gmez, C. (2009). Verifying Action Semantics Specifications in UML Behavioral Models. In *Advanced Information Systems Engineering*, volume 5565 of *LNCS*, pages 125–140. Springer Berlin / Heidelberg.

RTCA (2011a). RTCA Inc. DO-178C: Software Considerations in Airborne Systems and Equipment Certification.

RTCA (2011b). RTCA Inc. DO-331: Model-Based Development and Verification Supplement to DO-178C and DO-278A.

Rumbaugh, J. (1987). Relations as semantic constructs in an object-oriented language. In *ACM Sigplan Notices*, volume 22, pages 466–481. ACM.

Selic, B. (2012). The Less Well Known UML. In *Formal Methods for Model-Driven Engineering*, pages 1–20. Springer.

White, L. J. and Wiszniewski, B. (1991). Path testing of computer programs with loops using a tool for simple loop patterns. *Softw. Pract. Exper.*, 21(10):1075–1102.

Zhu, H., Hall, P. A., and May, J. H. (1997). Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427.