The Compliance Testing of Software Tools with respect to the UML standards specification - the ArgoUML case study

Panuchart Bunyakiati and Anthony Finkelstein
Dept. of Computer Science
University College London
London WC1E 6BT
United Kingdom
Email: p.bunyakiati@cs.ucl.ac.uk, a.finkelstein@cs.ucl.ac.uk

Abstract

In ICSE'08 we demonstrated the Java UML Lightweight Enumerator (JULE) tool, which supports compliance test generation from modeling standards specifications. When employed in our framework for software tool certification, JULE provides a powerful technology to enumerate a set of test cases that exhaustively test a modeling tool. JULE avoids combinatorial explosion by generating test cases only up to non-isomorphism. In this paper, a case study presented is an experiment on the use of a test suite generated from JULE to assess the compliance of an open source software tool - ArgoUML. This case study illustrates how ArgoUML is tested and reveals some previously unknown non-compliance issues. The case study highlights how software modeling tools can be tested for standards compliance and how test results can be analyzed to diagnose the causes of non-compliance in a software tool.

1. Compliance test generation with JULE

The Java UML Lightweight Enumerator (JULE) [4] tool provides automated support for compliance test generation focusing on the model analysis operations of software modeling tools with respect to the static semantics part of modeling language specifications. This case study describes compliance testing of ArgoUML against the Unified Modeling Language (UML) specification [14] and the Object Constraint Language (OCL) [15] well-formedness rules.

Compliance testing for software modeling tools is limited to experiments on the models upon which the software tools operate, to determine whether the models' conditions of compliance are maintained by the tools. The certification framework uses UML models as the test inputs for its approach based on bounded exhaustive-testing [18], the technique in which software is tested with all valid inputs up to a specified bound on the input size, and pseudo-exhaustive testing [13] where abstraction methods such as equivalence class are used to select test inputs.

From the UML metamodel and OCL well-formedness rules, the JULE tool enumerates a set of UML models up to non-isomorphism using the model generating technique described in [3]. Each member of this set is an exemplar of an equivalence class of models, within which structure is preserved but model element identities vary. Since OCL well-formedness rules are defined at the metamodel level, individual model element identities are not relevant.

2. Testing framework

In a compliance test suite, each test case is a pair consisting of a UML model and its condition of compliance (expected test result) that indicates whether the application satisfies or violates a particular well-formedness rule. This compliance test suite is classified into two categories of test data, demonstrations and counterexamples. The demonstrations are the set of valid models. They exist to detect the false-positive problems to ensure that the tools do not reject correct models. The counterexamples are the set of invalid models. They detect the false-negative problems in which the tools accept incorrect models. Fully compliant tools must accept all demonstrations and reject all counterexamples and testing a tool based on single examples from each equivalence class should reveal the majority of compliance errors.

To execute a test case, the software tool creates the test model and verifies it. The verification result is then compared with the expected test result to conclude a pass/fail compliance test result.

3. How JULE works

Test generation is performed by the four components of JULE: the OCL translator for processing OCL statements; the combinatorial package for generating the test data; Crocopat [2], a tool for relational computation based on Binary Decision Diagrams (BDDs), for creating expected test output; and JUnit [11] generator for producing test programs in Java.

Given an OCL well-formedness rule, JULE parses the well-formedness rule, constructs a test data specification for generating test and creates a Relational Manipulation Language (RML) [2] program for producing test oracle. A test data specification is a part of the UML metamodel and the number of objects for the metamodel types present. With this specification, JULE employs its combinatorial package to enumerate a set of non-isomorphic test cases, each of which is then submitted to Crocopat together with the RML program. The result returned is an expected test result which indicates whether the test case is a demonstration or a counterexample. Each pair of a test and an expected test result is concretized as a test in JUnit using the JUnit generator.

4. ArgoUML

ArgoUML [1] is a major open source UML modeling tool that supports the UML 1.4 standards specification and is available under the BSD license. This allows commercial tools such as Poseidon for UML [9] and MyEclipse UML [6] to extend from this open source project. The feature list of ArgoUML states that "ArgoUML is compliant with the OMG Standard for UML 1.4. The core model repository is an implementation of the Java Metadata Interface (JMI) which directly supports Meta Object Facility (MOF) and uses the machine readable version of the UML 1.4 specification provided by the OMG."

ArgoUML employs two methods for analyzing design models, first the design critics which analyze the models, suggest design improvements and indicate syntax and well-formedness errors and second, the preventive approach by embedding the well-formedness rule in methods for building a new model element. Before adding a new model element to the model, a build method is invoked to check whether the given parameters for building the new element are consistent with their relevant well-formedness rules. If the parameters are inconsistent with the rules, the method throws an exception indicating the problems. The source code was checked out from the ArgoUML repository at http://argouml.tigris.org/svn/argouml/ from release VERSION-0-26-ALPHA-1. Testing was conducted in the package org.argouml.model.mdr in the class CoreFactoryMDRImpl.java. The test cases were executed on a Pentium IV 1.50 GHz machine with memory 512 MB using JUnit3 in Eclipse 3.2 as a test runner. The sizes of the test suites range from 9 to 287 test cases. All tests were completed within 10 seconds and the test reports produced by JUnit give the list of test cases that were passed, failed or unfinished (errors). Using these reports the failures were identified and the causes of failures in the implementation were analyzed.

4.1 Non-compliance Issue I

The first experiment shows that even a short and uncomplicated well-formedness rule can be misinterpreted by programmers. The well-formedness rule for the AssociationEnd metaelement constrains that "the Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable away from that end." The OCL expression of this rule is shown below.

```
self.participant.oclIsKindOf (Interface) or
    self.participant.oclIsKingOf (DataType) implies
        self.association.connection->select(ae| ae <>
            self)->forAll(ae|ae.isNavigable = false)
```

Figure 1 the well-formedness rule for AssociationEnd

We used JULE to generate test cases within a bound to the input size of two AssociationEnds, one Association, three Classifiers, an Interface and a DataType. There were 27 test cases generated, 20 of them are demonstrations and 7 are counterexamples. Running these test cases in JUnit against ArgoUML found 2 failures that were both demonstrations. One of them was shown in figure 2 where the classifier of the context object is DataType and in the other failed test case, Interface. In both models, the other end of the association is not navigable, compliant with this well-formedness rule. However, ArgoUML reports that they are ill-formed. The implementation is over-constrained. By increasing the scope of the input size, the number of test cases increased accordingly. An example of these larger test cases is the one in figure 3. The test

results from the larger test suites were consistent with those of the smaller ones.

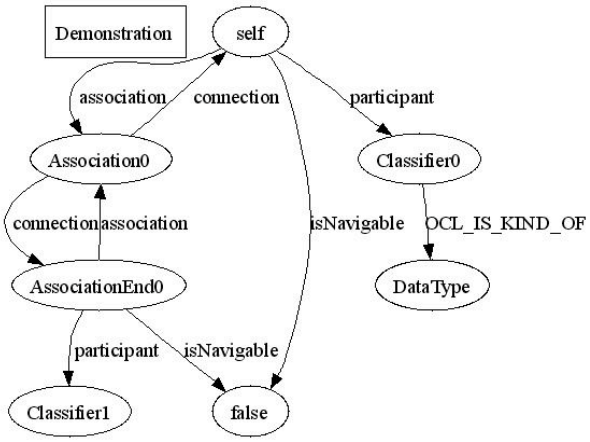

Figure 2. a test case for AssociationEnd

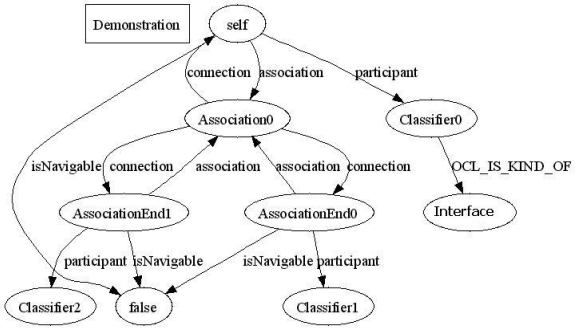

Figure 3. another test case for AssociationEnd at a larger scope

4.2 Diagnosis I

From the test results, a diagnosis can be made. ArgoUML rejects models whenever an end of the association has its participant of type either DataType or Interface that is not navigable. By running these failed test cases in Eclipse's debug mode, this diagnosis can be confirmed with the source code shown in figure 4. When the value of the variable type became an instance of DataType or Interface and the value of the variable navigable was false, the exception is thrown immediately. This confirms our initial diagnosis. The code snippet in figure 4 below - line 1 and 2 shows the errorneous conditions. The IllegalArgumentException was thrown from line 3-8.

```
1 if (type instanceof DataType || type instanceof Interface) {
2     if (!navigable) {
3           throw new IllegalArgumentException(
4                 "Wellformedness rule 2.5.3.3 [1] is broken. "
5                 +"The Classifier of an AssociationEnd cannot"
6                 +"be an Interface or a DataType if the "
7                 +"association is navigable away from "
8                 +"that end.");
9     }
10 List<AssociationEnd> ends = new ArrayList<AssociationEnd>();
11    ends.addAll(((UmlAssociation) assoc).getConnection());
12    for (AssociationEnd end : ends) {
13          if (end.isNavigable()) {
14          throw new IllegalArgumentException("type is either "
15            + "datatype or " + "interface and is "
16            + "navigable to");
17          }
18    }
19 }
```

Figure 4. code snippet from the buildAssociationEnd method

4.3 Non-compliance Issue II

The next problem uncovered was the second well-formedness rule applied to AssociationEnd. This rule states that "an instance may not belong by composition to more than one composite instance." The OCL statement of this well-formedness rule is shown in figure 5.

self.aggregation = composite implies
self.multiplicity.upperbound = 1

Figure 5. another well-formedness rule for AssociationEnd

For this rule, JULE generated only 9 test cases from one AssociationEnd, three AggregationKinds - Aggregate, Composite and None and three possible integer values: 0, 1 and 2. Because these values represent semantically different contexts, each combination of these values (the values of AggregationKinds and the integers) results in a semantically different model. The number of test cases is equivalent to the total number of Cartesian products of the two sets (3 possible aggregationKinds × 3 possible integers).

Testing ArgoUML with the 9 test cases reported 2 failures shown in figure 6(a) and 6(b). The two tests are the association ends that are composite and have upper bound 0 and 2 respectively. Clearly, both test cases are counterexamples; however, they went undetected.

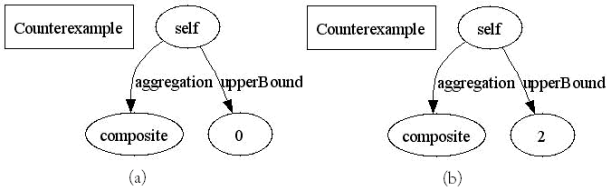

Figure 6. two test cases for AssociationEnd

4.4 Diagnosis II

Running these two test cases in Eclipse's debug mode found a problem in line 3 of code in figure 7 which always returns false no matter what the value of the variable multi is. Tracing to the getMaxUpper method discovered a fault - this method always returns 0. This can be fixed easily by changing line 9 of the code in figure 8 to return max and ArgoUML can detect all counterexamples correctly.

```
1 if (aggregation != null
2 &&aggregation.equals(AggregationKindEnum.AK_COMPOSITE)
3 && multi != null && getMaxUpper((Multiplicity) multi) > 1) {
4 throw new IllegalArgumentException("aggregation is composite "
5       + "and multiplicity > 1");
6 }
```

Figure 7. code snippet for the buildAssociationEnd method

```
1     private int getMaxUpper(Multiplicity m) {
2           int max = 0;
3           for (MultiplicityRange mr : m.getRange()) {
4                 int value = mr.getUpper();
5                 if (value > max) {
6                       max = value;
7                 }
8           }
9           return 0;
10    }
```

Figure 8. the getMaxUpper method

4.5 Non-complaince Issue III

The next issue was one of the rules that constrain the semantics of Generalization. This rule simply states that "Circular inheritance is not allowed." The OCL of this well-formedness rule is shown in figure 9. This rule excludes the self element from being in one of its allParents.

not self.allParents->includes(self)

Figure 9. a well-formedness rule for GeneralizableElement

The four test cases shown in figure 10 are counterexamples where self was involved, at some point, in circular inheritance. In the first model in figure 10(a), self is a child of itself. In the model in figure 10(b), self has a parent that is a child of itself through a generalizable element. In figure 10(c) and 10(d), self is a grandparent and great-grandparent of itself. All these models are invalid; however, ArgoUML can only detect the cases of circular inheritance in figure 10(b).

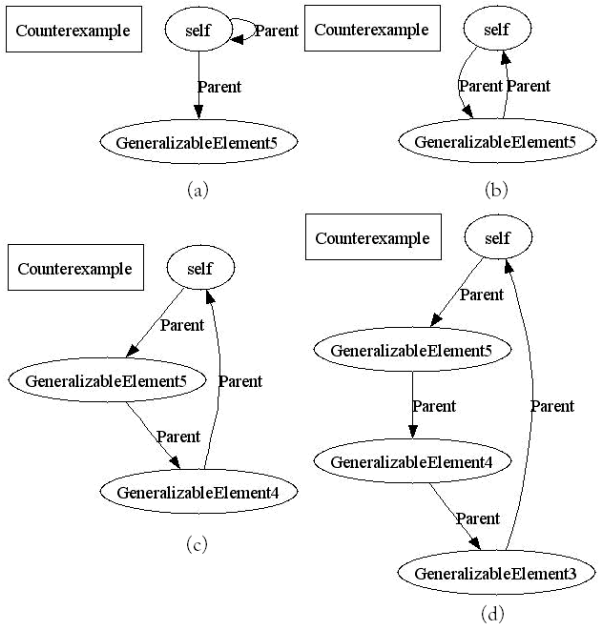

Figure 10. test cases for GeneralizableElement

4.6 Diagnosis III

The buildGeneralization method is shown in figure 11. The condition in line 5 should be "==" instead of "!=" - only when a child and its parent are the same object should the method throw an exception, not otherwise. The code in line 5 therefore can be changed to "|| (child1 == parent1)"

```
1   if((
2     !(child1 instanceof GeneralizableElement) ||
3     !(parent1 instanceof GeneralizableElement)
4     )
5     && child1 != parent1
6   ){
7     throw new IllegalArgumentException(
8     "Both items must be different generalizable elements");
9   }
```

Figure 11. code snippet from the buildGeneralization method

Next, consider the cases in figure 10(c) and 10(d), the grandchild and great grandchild circular inheritances. The code that handled these non-compliance issues was implemented in another part of the buildGeneralization method as shown in figure 12.

```
1   for (Generalization gen : parent.getGeneralization()) {
2     if (gen.getParent().equals(child)) {
3       throw new IllegalArgumentException("Generalization exists"
4           + " in opposite direction");
5     }
6 }
```

Figure 12. another part of code snippet from the buildGeneralization method

In line 1 of the code in figure 12, the body of the loop,

Generalization gen : parent.getGeneralization(),

takes all generalizations of the parent object. This is however incomplete, self.allParents is not limited to only the parents of the object from which it directly inherits, but according to the UML standards specification,

"the operation allParents returns a set containing all the generalizable elements inherited by this generalizable element (the transitive closure), excluding the GeneralizableElement itself."

The implementation in the buildGeneralization method deviates from this statement; this implementation only expresses the OCL below, but not equivalent to the original statement.

not self.parent.parents->includes(self)

Figure 13. a deviated well-formedness rule for GeneralizableElement

It was pointed out that circular generalization could be handled by one of the critics instead of by the build method. We tested ArgoUML with the model in figure 10(c) and 10(d) and found that there is a critic reporting problems in these models. With the previously mentioned correction, ArgoUML can deal with all four cases of circular inheritance correctly. It is then compliant with this well-formedness rule.

5. Related works

Farchi et al. [7] demonstrate test suite generation for parts of the POSIX standard and for the Java exception handling specification. Their method derives behavioral models from standards specifications. In contrast, JULE focuses on the static semantics part of modeling language specifications.

TestEra [12] uses a SAT solver in the Alloy Analyzer [10] to enumerate test models for checking the correctness of tools such as the fault-tree analyzer Galileo [5]. JULE limits its test generation differently and uses Binary Decision Diagrams (BDDs) implemented in Crocopat to check for model satisfactions.

6. Lessons learned

It is shown that black-box, bounded exhaustive testing using both demonstrations and counterexamples is a sound approach for compliance assessment for a software modeling tool. Some non-compliance issues can be detected by demonstrations and some by counterexamples. It can be said that this approach builds up the proof of compliance, within a boundary, using the proof-by-cases technique [16] where a proof is constructed on a case-by-case basis until all required cases are proved. Because test generation is bounded by the input size, it is up to the test engineers to decide when to stop the test.

As a general observation, we note that the approach of translating these well-formedness rules to Java code seems prone to error. It is possible that developers may misunderstand the well-formedness rules and implement them in Java incorrectly. Also, semantic variation points in the UML specification allow variations of model interpretation to support a variety of application domains. A more effective approach might be to implement a model validator that directly operates from OCL, as we have a formal semantics of this language. One implementation based on this approach is UCLUML [17].

7. Conclusion and future work

In this paper, we set out to test the feasibility of using the framework for software tool certification to a realistic software tool. The basis of this evaluation was an experiment on applying a test suite generated from JULE to the ArgoUML modeling tool. The results reveal three previously unknown faults in ArgoUML. The first issue was corrected by the ArgoUML team and removed from the current source code (revision 16250). We have reported the remaining issues to the ArgoUML development team. These issues have been corrected in revision 16693.

Because JULE supports test generation for modeling languages defined using EMOF/OCL, this, in principle, allows test generation for UML 1.4.2 as well as UML 2.x. We chose to experiment with UML 1.4.2

because it is recognized as an ISO standard - ISO/IEC 19501 and because of the availability of supporting modeling tools e.g. ArgoUML itself. The same principle applies for other domain specific languages (DSLs). For example, the Architecture Analysis & Design Language (AADL) [8] may be represented as a UML profile from which JULE may generate test directly.

As an immediate future work, we will also experiment this framework with commercial, non-open source tools to realize the impact of the unavailability of source code on the diagnosis of non-compliance issues.

8. Acknowledgement

The authors would like to thank Tom Morris from the ArgoUML team for his constructive feedback, James Skene for his contributions to the development of JULE and Andy Maule for his helpful suggestions.

9. References

[1] ArgoUML. Argouml. http://argouml.tigris.org/, 2005.

[2] D. Beyer. Relational programming with crocopat. In ICSE '06: Proceedings of the 28th ICSE, pages 807-810, New York, NY, USA, 2006. ACM.

[3] P. Bunyakiati, A. Finkelstein, and D. Rosenblum. The certification of software tools with respect to software standards. In IRI, pages 724–729. IEEE Systems, Man, and Cybernetics Society, 2007.

[4] P. Bunyakiati, A. Finkelstein, J. Skene, and C. Chapman. Using jule to generate a compliance test suite for the uml standard. In ICSE'08: Proc. of the 30th ICSE, pages 827–830, New York, NY, USA, 2008. ACM.

[5] D. Coppit and K. J. Sullivan. Galileo: A tool built from mass-market applications. In Proceedings of the 22nd ICSE, pages 750-3, Limerick, Ireland, 4-11 June 2000. IEEE.

[6] The Eclipse Project. The eclipse modelling framework (emf). http://www.eclipse.org/emf/.

[7] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. IBM Systems Journal, 41(1):89-110, 2002.

[8] P. Feiler, D. Gluch, J. Hudak, The Architecture Analysis & Design Language: An Introduction", SEI, Technical Note CMU/SEI-2006-TN-011

[9] Gentleware A. B. Poseidon uml editor. http://www.gentleware.com/.

[10] D. Jackson, Software Abstractions: Logic, Language and Analysis. MIT Press. Cambridge, MA.

[11] JUnit, November 2007, DOI=http://www.junit.org/

[12] Khurshid, S. and Marinov, D. TestEra: Specification-Based Testing of Java Programs Using SAT. Automated Software Engg. 11, 4 (Oct. 2004), 403-434.

[13] D. R. Kuhn and V. Okum. Pseudo-exhaustive testing for software. Proc. 30th NASA/IEEE Software Eng. Workshop, 2006, pages 153–158.

[14] Object Management Group. OMG Unified Modeling Language, Version 1.4. OMG, http://www.omg.com/uml/, 2001.

[15] Object Management Group. UML 2.0 Object Constraint Language (OCL) Specification. Object Management Group, 2003. http://www.omg.org/docs/ptc/03-10-14.pdf.

[16] M. Pezze, M. Young, Software Testing and Analysis: Process, Principles and Techniques, Wiley, 2007.

[17] J. Skene. The ucl mda tools.

http://uclmda.sourceforge.net/index.html, 2007.

[18] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. IEEE Transactions on Software Engineering, VOL. 31, NO. 4, APRIL 2005 pages 133–142, 2004.