Using JULE to Generate a Compliance Test Suite for the UML Standard

Panuchart Bunyakiati, Anthony Finkelstein, James Skene and Clovis Chapman
Dept. of Computer Science, University College London
London WC1E 6BT
United Kingdom
{P.Bunyakiati, A.Finkelstein, J.Skene, C.Chapman}@cs.ucl.ac.uk

ABSTRACT

The Java-UML Lightweight Enumerator (JULE) tool implements a vitally important aspect of the framework for software tool certification - test suite generation. The framework uses UML models as the test inputs for the bounded exhaustive-testing approach. Within a size bound for the metamodel types, JULE enumerates only the set of non-isomorphic models in the form of relational structures. These models are classified into two sets - demonstration and counterexample - using Binary Decision Diagrams (BDDs). The power of JULE lies in its model enumeration and its use of a high-performance grid infrastructure. Hence, JULE efficiently generates a very small test suite while increasing the bound on the input size to the extent that is practical for certification purpose.

1. COMPLIANCE TEST GENERATION

An outstanding issue today in the software tools industry is the standards compliance of software tools required to support interoperability. To assess this compliance, the JULE tool aims to provide automated support for compliance test generation focusing on the model analysis operations of software modeling tools.

JULE implements compliance-test generation for tools using the Unified Modeling Language (UML), from Object-Constraint Language (OCL) [14] well-formedness rules embedded in the UML standard [15]. In our framework for software tool certification [4], compliance testing is limited to experimentation on the work products upon which the software tools operate to determine whether conditions of compliance are maintained by the tools.

The certification framework uses UML models as the test inputs for its bounded exhaustive-testing approach [17]. A compliance test case is a pair of a UML model and a test oracle stating whether the model satisfies or violates a particular well-formedness rule. For each compliance test case, the software tool creates the test model and verifies it. The verification result is compared with the expected result in the test oracle to conclude a pass/fail compliance test result.

This compliance test suite is composed of two categories of test data i.e. demonstrations and counterexamples.

Copyright is held by the author/owner(s).

ICSE'08, May 10–18, 2008, Leipzig, Germany.

ACM 978-1-60558-079-1/08/05.

The demonstrations are the set of valid models. They exist to detect the false-positive non-compliance and to ensure that the tools do not reject correct models. The counterexamples are the set of invalid models. They detect the false-negative non-compliance in which the tools accept incorrect models. Fully compliant tools must accept all demonstrations and reject all counterexamples.

For a given part of the UML metamodel, it is possible to generate all model configurations within a finite number of instances for the metamodel types present. However, the number of models increases rapidly due to combinatorial explosion. Therefore JULE employs the model generating technique described in [4] to generate only the set of non-isomorphic models, each member of which 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, and testing a tool based on single examples from each equivalence class should reveal the majority of compliance errors.

2. EXAMPLE

To illustrate how JULE works, we provide two examples from the well-formedness rules of the Association model element. For each example, we show the relevant part of the UML metamodel, the well-formedness rules under test and two test cases, a demonstration and a counterexample. We then present for each test the number of total models possible for stated size bounds, the number of equivalence classes within those models and within the equivalence classes the numbers of demonstrations and counterexamples.

Example I

The first example shows that test cases can be generated succinctly for a small number of instances. We test the well-formedness rule (2.1) which constrains association ends to have a unique name within the association. Multiplicity constraints in the metamodel require that (1) an association must have two or more association ends as its connections and (2) an association end must have exactly one string as its name.

ModelElement	Attribute	Type	Multiplicity
Association	Connection	AssociationEnd	2..n
AssociationEnd	Name	String	1

(Rule 2.1) self.allConnections()->forAll(r1, r2 |
r1.name = r2.name implies r1 = r2)

Figure 1: the metamodel and well-formedness rule of the Association model element

Table 1 lists the numbers of possible models, test cases, demonstrations and counterexamples for models having a single association, *n* association ends and *n* strings.

Table 1: the size of the test suite for rule 2.1

n	Total	Test cases	Demonstration	Counterexample
2	4	2	1	1
3	108	6	3	3
4	2816	15	7	8
5	81250	28	13	15
6	2659392	55	24	31
7	98825160	90	39	51
8	4.14e10	154	64	90
9	1.94e11	240	98	142
10	1.01e13	378	150	228
11	5.80e14	560	219	341
12	3.64e16	847	322	525

For n=3, three sets of model elements (referred to as model element domains) are created including a singleton set for association model element i.e. self, a set of three association end model elements - associationend0, associationend1 and associationend2 and a set of three name model elements -name0, name1 and name2. Out of 108 possible models that can be constructed by assigning elements in a model element domain as attributes of other elements in another domain according to metamodel, JULE generates only six non-isomorphic models. Classifying these non-isomorphic models results in three demonstrations which each association end of an association has its own unique name. The other three models are counterexamples in which two or more distinct association ends of an association share a common name. This non-isomorphic generation significantly reduces the size of the test suite. When we increase the size to, for example, seven, the size of the test suite may be reduced from 98 million structures to only 90.

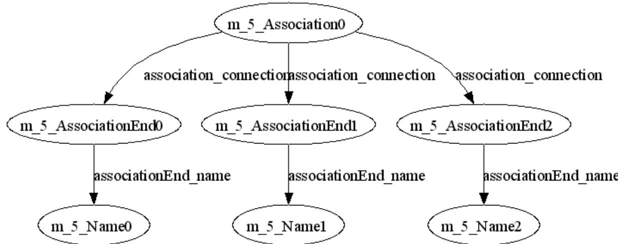

Figure 2: a demonstration for rule 2.1

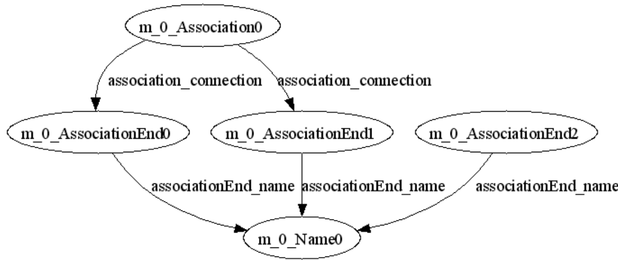

Figure 3: a counterexample for rule 2.1

Example II

The second example shows that it is possible to generate a test suite for set of rules that constrain the same set of model elements. The aggregation attribute of an association end specifies whether the instance on the association end is an aggregation. Three possibilities are that, the instance is an aggregate, a composite or a part. The sizes of the instances may be varied but there will be a fix number of aggregation kinds, which is three i.e. aggregate, composite and none. For example, consider the rule (2.2) stating that "at most one AssociationEnd may be an aggregation or composition" and the rule (2.3) "if an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition."

ModelElement	Attribute	Type	Multiplicity
Association	Connection	AssociationEnd	2..n
AssociationEnd	aggregation	AggregationKind	1

(Rule 2.2) self.allConnections()->select
(aggregation=AggregationKind.aggregate or
aggregation = AggregationKind.composite)->size() <= 1
(Rule 2.3) self.allConnections()->size() >= 3 implies
self.allConnections()->forAll
(aggregation = AggregationKind.none)

Figure 4: the metamodel and well-formedness rules for the Association model element

Out of 108 possible configurations that can be constructed from an association and three association ends, JULE generates 20 non-isomorphic models, 14 models more than those of the first example. This is due to the fact that association kind of the association ends is now significant, limiting structural equivalence.

Table 2: the size of the test suite for rule 2.2 and 2.3

N	Total	Test cases	a	b	c	d	e	f
2	9	6	3	3	6	0	3	3
3	108	20	8	12	11	9	6	14
4	891	45	13	32	17	28	9	36
5	6318	84	18	66	24	60	12	72
6	41553	140	23	117	32	108	15	125
7	262440	216	28	188	41	175	18	198
8	1620567	315	33	282	51	264	21	294
9	9880866	440	38	402	62	378	24	416
10	59816637	594	43	551	74	520	27	567
11	3.60e8	780	48	732	87	693	30	750
12	2.16e9	1001	53	940	101	900	33	968

Column: a – Rule 2.2 demonstrations b – Rule 2.2 counterexamples, c – Rule 2.3 demonstrations, d – Rule 2.3 counterexamples, e – Rule 2.2 and 2.3 demonstrations, f – Rule 2.2 and 2.3 counterexamples

In Figure 5, one association end is an aggregation. While in Figure 6, the association has three association ends and none of them is an aggregation or composition. Therefore, both models are demonstrations. In Figure 7, one of three association ends is an aggregation which is not allowed; therefore, this model is a counterexample.

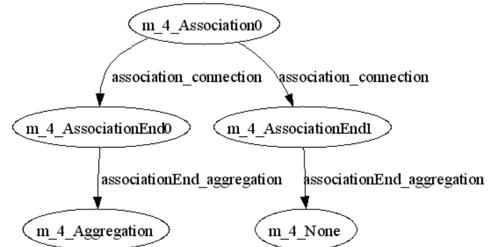

Figure 5: a demonstration for the rule 2.2 and 2.3

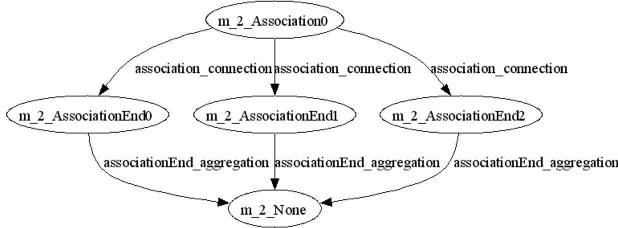

Figure 6: another demonstration for rule 2.2 and 2.3

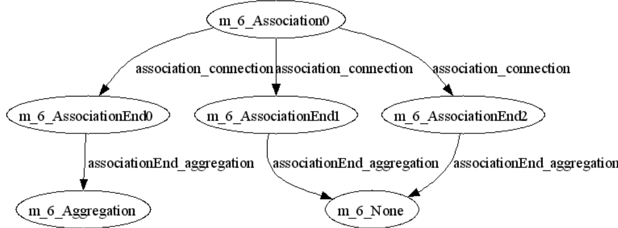

Figure 7: a counterexample for the rule 2.2 and 2.3

3. HOW JULE WORKS

JULE is an OCL language processor built on top of the UCLUML repository [16]. Given an OCL statement, JULE constructs a Java program that generates the test data and a Relational Manipulation Language (RML) program [2] that produces the test oracle. This process is performed by the four components of JULE: the OCL translator; the combinatorial package; Crocopat [2] a tool for relational computation based on BDDs; and JUnit [11] generator as described below.

3.1 Translating the well-formedness rules

The parser implemented in JULE constructs syntax trees from OCL statements based on the OCL metamodel. The results of the parsing are annotated syntax trees that have the instances of the OCL metamodel as the nodes and the terms of the parsed OCL statement as the attribute values. These annotated syntax trees reference the structure of the metamodel stored in the UCLUML repository. This information includes the types, relationships and multiplicity constraints present in the UML metamodel.

Table 3: the OCL-RML translation examples

<table>
<tr><th>OCLExpression</th><th>OCL</th><th>RML</th></tr>
<tr><td>PropertyCallExp</td><td>self.name</td><td>name(self,X)</td></tr>
<tr><td>OperationCallExp</td><td>self.allConnections()</td><td>allConnections(self,X)</td></tr>
<tr><td>LibraryOperation-CallExp</td><td>self.name = 'tyger'</td><td>name(self,X) & @"tyger"(X)</td></tr>
<tr><td>LibraryOperation-CallExp</td><td>self.name = 'tyger' implies not(self.name = 'tigger')</td><td>Name(self,X) & @"tyger"(X) -> !(name(self,X) & @"tigger"(X))</td></tr>
<tr><td>IteratorExp</td><td>self.allConnections()->forAll (r | r.name = self.name)</td><td>FA(r,allConnections(self,r) -> (name(r,X) & name(self,Y) & =(X,Y)))</td></tr>
<tr><td>IteratorExp</td><td>self.allConnections()->select (r | r.name = self.name)</td><td>allConnections(self,r) & name(r,X) & name(self,Y) & =(X,Y)</td></tr>
</table>

JULE then produces the Java program for enumerating test data according to this information. Also, JULE systematically translates the OCL statement to an RML program using translation rules; examples are shown in Table 3. These rules recursively replace the sub-trees of the OCL syntax trees with the groups of the RML nodes that are semantically equivalent to them.

3.2 Enumerating the test data

The test generating program invokes the combinatorial package to enumerate the models of all possible configurations under the bound to the given instance size. The combinatorial package implements algorithm for partition-multiplication that produces only non-isomorphic models. In essence, this algorithm for partition-multiplication performs in two steps: the partitioning of each relationship and then the multiplication of the partitions. The partitioning is the generation and selection of a non-isomorphic subset of the power set of E, where E is the set of the edges in the complete bipartite graph between the two sets of related model elements. The multiplication produces the Cartesian product of the sets of partitions results in the partitioning step. This product is the complete set of non-isomorphic test input.

3.3 Classifying the relational structures

JULE produces the test oracles by solving the satisfaction of the models to the well-formedness rules. The models and the RML program are submitted to Crocopat which returns the results indicate whether a model is a demonstration or a counterexample. As the test input can become very large, JULE slices this set of test input into hundreds of parts, each part contains only one model, and schedules them to the UCL Condor pool [5], a distributed job scheduling and resource management system. This helps producing the test oracle very quickly even for a very large number of test inputs. This technique is of course reliant on the availability of a large number of computational resources.

3.4 Testing

JULE uses JUnit, a framework for automating unit testing, to execute the test suite. JULE displays each test case as a graph generated using dot [8], and in Rigi Standard Format (RSF) [18] and generates the JUnit test files that invoke the API of the tool under test to create the test model, invoke the verification method of the tool under test, compare the verification result with the test oracle and report the test results. JULE requires a user to provide the body of the model constructing methods such as *createClassifier()* and *setConnection(Association a, AssociationEnd e)* which differ from one tool to another.

4. RELATED WORKS

4.1 Light-weight formal method tools

Tools such as USE [9], Alloy Analyzer [10] and the VDM-SL Toolbox [7] may be used to analyze the specification of software systems. These tools are light-weight because they are not designed for proving the correctness or analyzing the soundness and completeness of the specification. But relying on the small-scope hypothesis [10], these tools rather support the assertion of the specifications by finding a model that satisfy the constraints ensuring that legal states are not completely ruled out or finding a counterexample to reveal flaws in the specifications. These tools are not testing tools; however, JULE relies on the same small-scope hypothesis.

4.2 Model-based testing

The idea of using model finding to generate test cases is not novel. TestEra [12] claims the contribution of using SAT solvers in the Alloy Analyzer to enumerate test data. TestEra and Korat [3] use pre/post models to generate test data from the precondition and use the post condition as a test oracle. In TestEra, the test data is reduced by breaking the symmetry. Korat also prunes the search effectively by monitoring the accesses to all the fields of the candidate input. In contrast, JULE limits its test generation differently, and is capable of identifying false-negative non-compliance.

4.3 Test generation by DNF partitioning

The technique by which the specification is partitioned into disjunctive normal form (DNF) to generate test cases is given in [6], which focuses on the implementation of a tool. Later, a more theoretical work supporting the idea is given in [13]. Recently, [1] takes a subset of OCL and uses a constraint solver to generate test cases for mutation-testing which requires prior knowledge about false patterns. Also, this implementation can not test OCL constraints including quantifiers.

5. LEVEL OF MATURITY

The development of JULE was initiated in January 2007 and is under active development. It provides support for generating the test suite for the well-formedness rules in the Foundation::Core package of UML 1.4.2 used in testing the UCLUML and ArgoUML tools. JULE aims to support test generation for modeling languages defined using EMOF/OCL Beyond our current work we believe JULE is likely to be helpful in generating test cases for analysis tools of other modeling languages.

6. CONCLUSION

This paper describes the application of JULE for generating a test suite for the UML standard. Given the UML metamodel and OCL well-formedness rules, JULE generates a set of demonstration and counterexample test cases which have UML models as test input. To illustrate the use of JULE, two examples of tests on association model elements are provided.

7. ACKNOWLEDGEMENT

The authors would like to thank Andrew Dingwall-Smith for his early contributions to the model enumeration algorithm and Andy Maule for his helpful suggestions concerning RML programming. We are grateful to the reviewers for their constructive suggestions and to the University of the Thai Chamber of Commerce for their funding of Panuchart Bunyakiati.

8. References

[1] Aichernig, B. K., Salas, P. A. P. 2005. Test Case Generation by OCL Mutation and Constraint Solving. In Proceedings of the QSIC. 64-71.

[2] Beyer, D. 2006 Relational Programming with CrocoPat. In Proceedings of the ICSE, Shanghai, China, 807-810.

[3] Boyapati, C., Khurshid, S. and Marinov, D. 2002. Korat: Automated Testing Based on Java Predicates, In Proceedings of the ISSTA, Rome, Italy.

[4] Bunyakiati, P., Finkelstein, A. and Rosenblum, D. 2007. The Certification of Software Tools with respect to Software Standards. In Proceedings of the IEEE IRI, Las Vegas, USA, 724-729.

[5] Chapman, C, Goonatilake, C., Emmerich, W., Farrellee, M., Tannenbaum, T., Livny, M., Calleja, M. and Dove, M. 2005. Condor Birdbath - Web Service interface to Condor. In Proceedings of the UK E-Science All Hands Meeting, Nottingham, UK.

[6] Dick, J. and Faivre, A. "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications", LNCS, vol. 670, Springer-Verlag, London. 268-284.

[7] Fitzgerald, J. and Larsen, P. G. 1998. Modelling Systems: Practical Tools and Techniques in Software Development, Cambridge Uni. Press.

[8] Gansner, E.R. and North, S.C. 2000. "An Open Graph Visualization System and Its Applications to Software Engineering," Software---Practice and Experience, v. 30, n. 11, 1203-1233.

[9] Gogolla, M., Bohling, J. and Richters, M. 2003. Validation of UML and OCL Models by Automatic Snapshot Generation. In Proceedings of the UML'2003, Lecture Notes in Computer Science, Vol. 2863, Springer, Berlin.

[10] Jackson, D. 2006. 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. 2001. TestEra, "A novel framework for automated testing of Java programs", In Proceedings of the IEEE ASE, CA, USA.

[13] Meudec, C. 1998 Automatic Generation of Software Test Cases from Formal Specifications. Doctoral Thesis, the Queen's University of Belfast.

[14] The Object Management Group (OMG), The Object Constraint Language (OCL) specification, DOI=http://www.omg.org/.

[15] The Object Management Group (OMG), The Unified Modeling Language (UML) specification, DOI=http://www.omg.org/.

[16] Skene, J., and Emmerich, W. 2006. Specifications, not Meta-Models. In Proceedings of the ICSE Workshop on Global integrated Model Management, China, 47-54.

[17] Sullivan, K., Yang, J., Coppit, D., Khurshid, S., and Jackson, D. 2004. Software assurance by bounded exhaustive testing. In Proceedings of the ISSTA '04. ACM, New York, NY, 133-142.

[18] Wong, K. 1998. The Rigi User's Manual - Version 5.4.4. DOI=http://www.rigi.cs.uvic.ca/downloads/rigi/doc/