www.ietdl.org

Published in IET Software
Received on 26th March 2010
Revised on 20th September 2010
doi: 10.1049/iet-sen.2010.0032

Special Section: Automation of Software Test (AST '09)

ISSN 1751-8806

Standards compliance testing for unified modelling language tools

P. Bunyakiati[1] *A. Finkelstein*[2]

[1]*School of Science, University of the Thai Chamber of Commerce, Bangkok 10400, Thailand*
[2]*Department of Computer Science, University College London, London WC1E 6BT, UK*
E-mail: panuchart_bun@utcc.ac.th

Abstract: Software modelling standards such as the unified modelling language (UML) provide complex visual languages for producing the artefacts of software systems. Software tools support the production of these artefacts by providing model constructs and their usage rules. Owing to the size and complexity of these standards specifications, establishing the compliance of software modelling tools to the standards can be difficult. As a result, many software tools that advertise standards compliance may fail to live up to their claims. This study presents a compliance testing framework to determine the conditions of compliance of tools and to diagnose the causes of non-compliance issues. The Java-UML lightweight enumerator (JULE) tool realises this framework by providing a powerful technology to create a compliance test suite for modelling tools. JULE generates test cases only up to non-isomorphism to avoid combinatorial explosion. An experiment with respect to the UML 1.4 is presented in this study. The authors test ArgoUML for its compliance with the UML 1.4 specification. The authors also report some findings on four UML 2.x tools, including Eclipse Galileo UML2, Enterprise Architect 7.5, Poseidon for UML 8.0 and MagicDraw 16.6.

1 Software development standards and compliance assessment of software tools

Software modelling standards such as the unified modelling language (UML) provide complex modelling languages for producing artefacts of software systems. Software tools enact software development processes, automate activities and support the production of artefacts defined in the standards. An important issue today in the software tools industry is interoperability. Standards compliance enables interoperability and allows software modelling tools to interchange software artefacts. However, establishing standards compliance in software tools can be difficult. Some preliminary findings [1] show that many existing tools advertising standards compliance fail to live up to their claim.

In addition, determining the conditions of standards compliance of these tools is not trivial, owing to the size and complexity of the standards specifications. Because software standards are defined using many model constructs and usage rules, generating correct and complete compliance test suites for these standards specifications is not a straightforward task. The difficulty arises from two main reasons. Firstly, many test cases may not be considered valid because they are not appropriate to the usage rules of the language under consideration, that is, it is unnecessary to test a large number of invalid test cases that are impossible to occur in reality. Secondly, despite the fact that it is possible to generate all test cases within the given bounds on the number of the model elements present, when the size bounds increase, the number of test cases increases rapidly because of combinatorial explosion. This limits the test coverage to a very small number of model elements.

It can be said that compliance testing for software tools supporting other domain-specific languages (DSLs) also suffers from similar problems. Because our framework present here supports test generation for modelling languages defined using EMOF/OCL [2, 3], this, in principle, allows test generation for other DSLs such as the architecture analysis and design language (AADL) [4] that can be represented as a UML profile from which our technique may generate test directly.

In this paper, we propose a novel framework to determine standards compliance level of UML modelling tools. This framework is realised in the Java-UML lightweight enumerator (JULE) tool [5, 6] that provides automated support for compliance test suites generation focusing on the model analysis operations of software modelling tools. Our compliance testing is limited to experiments on the work products on which the software tools operate to determine whether conditions of compliance are maintained by the tools. These experiments are conducted on a case-by-case basis. Each test case is a pair of a work product, here – a software model – and its condition of compliance indicating whether the model satisfies or violates some constraints defined in the standards specification.

In general, the model analysis operation is a Boolean function. It takes a software model as an input and returns a Boolean value as an output – TRUE if the model satisfies the constraints and FALSE otherwise. Therefore there are two types of non-compliance errors that could possibly

© The Institution of Engineering and Technology 2011

exist. The first type of error occurs when a software tool rejects a valid model (returns FALSE instead of TRUE). The second type occurs when a software tool accepts an invalid model (returns TRUE instead of FALSE).

A test suite requires two categories of test cases to be considered sufficient – demonstrations and counterexamples. The demonstrations are the set of valid models employed to ensure that a software tool does not reject well-formed models. The counterexamples are the set of invalid models used to check that a software tool does not accept ill-formed models. Fully compliant tools must accept all demonstrations and reject all counterexamples in the test suite.

2 Compliance testing for the UML specification

This work focuses on one of the most important software modelling standards – the UML [7] – which has become the de facto standard for object-oriented software modelling. The UML is a large document of technical specifications. A great deal of the specification is the abstract syntax defined with the metamodelling approach using meta object facility (MOF) [2]. MOF is the meta-meta model used to describe the UML standards specification – the UML metamodel. This UML metamodel consists of a set of metaelements that are used to create model elements in a UML model. Each model element in a UML model is an instance of a metaelement in the UML metamodel. Software tools support the production of software systems by providing a functionality to create instances of these metaelements such as Classifier and Association. Software tools bridge the gap between the UML metamodel and its models. For any UML model created with the tools, the properties of the metaelements must be preserved; that is, the one-to-one correspondence between the metaelements provided in the UML tools and those specified in the UML metamodel must be established.

In addition to the UML metamodel, an important feature of the UML specification is the object constraint language (OCL) [3]. The OCL can enhance the expressive power of the UML. Formal constraints or 'well-formedness rules' are specified as part of the language specification to restrict the construction of UML models. These rules allow the models built according to the UML specification to be verified for their well formedness. A subset of the well-formedness rules in the UML specification is used in the case studies to check that the core constructs of the specification are implemented in the tools correctly. This subset includes the rules that govern the usage of Association, Composition, Aggregation and Generalisation in the UML class diagram.

To illustrate this, Fig. 1 shows a well-formedness rule and its relevant parts of the UML metamodel described in the UML 2.2 specification. This rule enforces the number of member ends of a specialising association to be the same as those of its generalising association. This rule requires, at the minimum, two test cases as presented in Fig. 2. The demonstration on the left shows an association A0 connecting to two member ends – a and b. In addition, A0 is specialised by another association A1 which connects to the same number of member ends – c and d. In contrast, the counterexample on the right shows an association B0 connecting to two member ends – e and f, but B0 is specialised by the association B1 that connects to three member ends – g, h and i – instead of only two.

To execute the test cases in Fig. 2, a software tool creates the test model of each of the test cases and then verifies it.

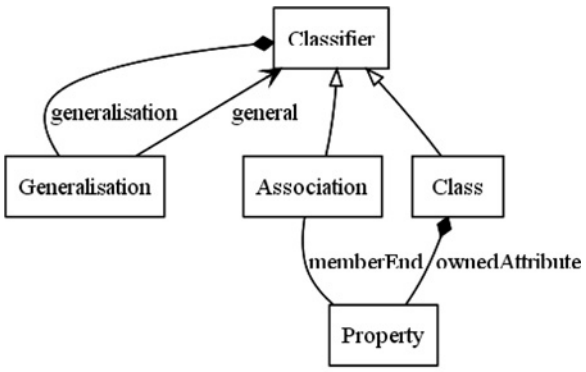

Fig. 1 *Subset of the core package of the UML 2.2 specification*

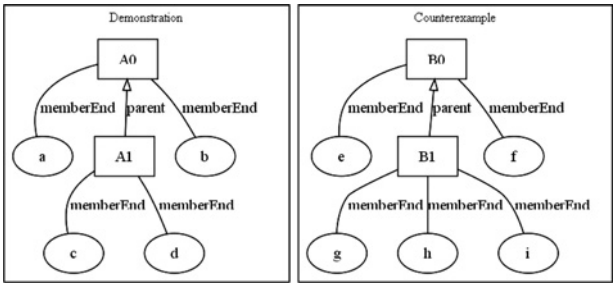

Fig. 2 *Demonstration and a counterexample for [rule 1]*

The result of this verification, also called actual result, is then compared with the expected result to conclude a pass/fail compliance test result. A compliant tool must return TRUE when validating the demonstration and FALSE when validating the counterexample.

3 Pseudo-exhaustive compliance testing

The heart of compliance test generation is to automatically generate only necessary models and to avoid generating unnecessary ones. Exhaustively generating all models can suffer from combinatorial explosion. To effectively test for compliance, it is important to generate only the set of non-isomorphic models, each member of which is an exemplar of an equivalence class of model configurations, within which structure is preserved but the identities of the model elements vary. Because OCL well-formedness rules are defined at the metamodel level, individual model-element identities are not relevant.

Consider another example from the UML 1.4 standards specification in Fig. 3, an association must have at least two association ends. Each association end must have exactly one string as its name. The well-formedness rule 2 constrains each association end within an association to have a unique name. The two models 4a and 4b in Fig. 4 are generated with respect to this specification. Both the models have one end of association with a unique name

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

[rule 2] each association end within an association must have a unique name
self.allConnections()->forAll(r1, r2 | r1.name = r2.name implies r1 = r2)

Fig. 3 *Metamodel and well-formedness rule of association*

© The Institution of Engineering and Technology 2011

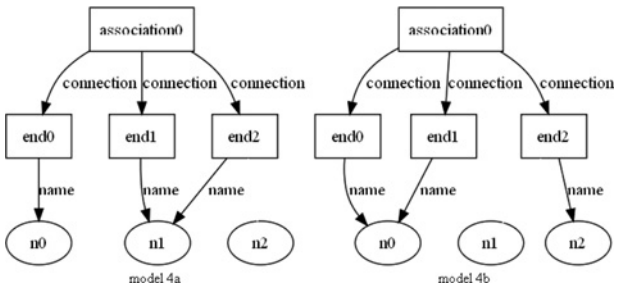

Fig. 4 *Two isomorphic models – 4a and 4b*

and two other ends sharing the same name. Clearly, 4a and 4b are isomorphic and are in the same equivalence class. Only one of the two models is required to be included in the test suite.

Formally, let E_c be the set of model elements of type c and suppose there are t types of metaelements present – namely $c_0, \ldots, c_t$ – a model M consists of some model elements from E where $E = E_{c0} \cup \ldots \cup E_{ct}$ – and let $L(M)$ be the set of links of M, together with a function ν_M which associates with each link a pair of model elements in M. Two models M_0 and M_1 are said to be isomorphic if there is a bijection e: $E_c(M_0) \rightarrow E_c(M_1)$ and ζ: $L(M_0) \rightarrow L(M_1)$, where each link $\nu_{M0}(l) = st$ in M_0 is preserved by $\nu_{M1}(\zeta(l)) = e(s)e(t)$ in $M1$.

For a small number of model elements, non-isomorphic test case generation can produce a succinct test suite. When the number of model elements increase, non-isomorphic test case generation can reduce the size of the test suite significantly. Fig. 5 shows the numbers of model-elements n increasing from two to seven, total configurations, test cases and the classification of test cases into demonstrations and counterexamples. For $n = 2$, the model-elements include a single association, two association ends and two strings, out of four total configurations only two test cases are generated. For $n = 7$, there are over 98 million total configurations but the size of test suite may be reduced to only 90 test cases.

4 How JULE works

Test generation is performed by the four components of JULE depicted in Fig. 6: the OCL translator for processing OCL statements (compilation); the combinatorial package for generating the test data (enumeration); Crocopat [8], a tool for relational computation based on binary decision diagrams (BDDs) [9] that is a compact representation of Boolean expressions, for creating expected test output (classification); and JUnit [10] generator, for producing test programs in Java (test case generation).

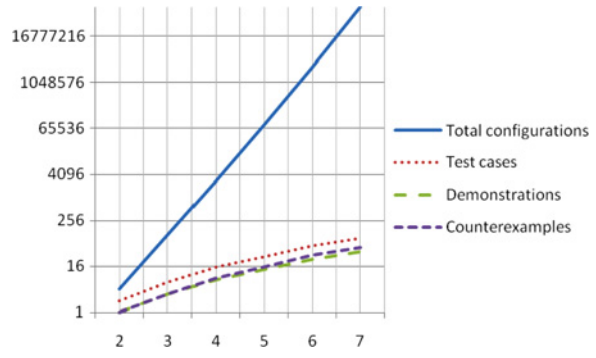

Fig. 5 *Size of test suite for well-formedness rule 2*

Given an OCL well-formedness rule, JULE parses the rule, constructs a test data specification for generating test and creates a relational manipulation language (RML) [8] program for producing test oracle. Some examples of this translation are shown in Table 1.

A test data specification is a part of the UML metamodel and the number of model elements for the metamodel types present. With this specification, JULE employs its combinatorial package to enumerate a set of non-isomorphic test cases. This activity is described in detail in the Section 5. Each of the test cases 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. Section 6 describes the process of test classification and how each pair of test model and its expected test result is concretised as a JUnit test case. An example of the JUnit test case is shown below in Fig. 7.

5 Partition–graphicalisation–multiplication method

This section elaborates the enumeration and classification components discussed in Section 4. After processing the UML metamodel and a well-formedness rule, the partition–graphicalisation–multiplication method, implemented in the combinatorial package of JULE, is used to generate a set of non-isomorphic models from a finite set of model elements of types present in the metamodel. Then, the output of this method is classified into two sets, demonstrations and counterexamples using an implementation of BDDs, Crocopat.

The partition–graphicalisation–multiplication method consists of three steps that is, partition, graphicalisation and multiplication. First the method enumerates integer–partition pairs for each relationship present in the well-formedness rule according to the multiplicity constraints of that relationship. From the integer–partition pairs, the method generates non-isomorphic bipartite graphs based on the principle that two bipartite graphs generated from two different partition pairs are guaranteed to be non-isomorphic because the degrees of the nodes in the graphs are always different. Finally, the bipartite graphs from each relationship are combined with those of other relationships in the same well-formedness rule to construct the test models.

5.1 Partition

To enumerate test models within a given number of model elements, the partition step must consider the number of model elements of each type together with multiplicity

Fig. 6 *Components of JULE and the test generation process*

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2010.0032

Table 1 Some translations from OCL statements to RML programs

OCL statements	RML programs
a.name	name(a,X)
a.memberEnd	memberEnd(a,X)
a.memberEnd.size()	#(memberEnd(a,X))
self.parents()- > forAll(p|	FA(p,parent(self,p)->
p.memberEnd.size() = self.memberEnd.size())	#(memberEnd(p,X)) = #(memberEnd(self,Y)))

```
public void testAssociation_1_2_0() throws Exception {
        Object self0 = Association_0Helper.createAssociation();
        Object a0 = Association_0Helper.createAssociation();
        Object AssociationEnd0 = Association_0Helper.createAssociationEnd();
        Object AssociationEnd1 = Association_0Helper.createAssociationEnd();
        Association_0Helper.setAssociationMember(self0,AssociationEnd0);
        Association_0Helper.setAssociationMember(self0,AssociationEnd1);
        Association_0Helper.setAssociationMember(a0,AssociationEnd0);
        Association_0Helper.setAssociationMember(a0,AssociationEnd1);
        Association_0Helper.setGeneralisation(a0,self0);
        boolean satisfied = Association_0Helper.validate( self0, AssociationRule_1);
        assertEquals(satisfied, true);
}
```

Fig. 7 *Code snipet from a JUnit test case for the well-formedness rule*

constraints to assign each model element its possible in- and out-degrees of connections. The result of this step is called the integer–partition pairs. The definitions of integer partitions, restricted integer partitions and integer–partition pairs are introduced as follows.

Definition 1: Integer partition – an integer partition $P(q) = [p_0, p_1, \ldots, p_n]$ is defined as a list of integers where $q = \sum_{i=0}^{n} p_i$. For instance, $P(3) =$ [3], [2, 1] and [1, 1, 1].

Definition 2: Restricted integer partition – the integer partition may be extended to a restricted integer partition $P(q, l, u, m, n)$ where $q = \sum_{i=0}^{r} p_i$ such that $m - 1 \leq r \leq n - 1$ and $l \leq p_i \leq u$. For example, $P(3, 1, 3, 2, 3) =$ [2, 1] and [1, 1, 1].

Definition 3: Integer–partition pair $\rho(S, T, q, l, u, v, w)$ – given two domains of model elements S and T, a number of edges q and a multiplicity constraint $\{l, u, v, w\}$, a partition pairs ρ represents a pair of restricted integer partitions $[P_s(q, l, u, 0, |S|), P_t(q, v, w, 0, |T|)]$. For instance, given the domain $S = \{s0\}$, $T = \{t0, t1, t2\}$, where $q = 3$ and a multiplicity constraint $\{2, 3, 1, 1\}$, the partition pairs of (S, T, q, 2, 3, 1, 1) is [2][1, 1] and [3][1, 1, 1].

To illustrate this, consider again the well-formedness rule 2 in Fig. 3. Where $n = 3$, it follows that the domain Association = {self}, AssociationEnd = {end0, end1, end2} and String = {$n0$, $n1$, $n2$}. The multiplicity constraint of the relationship Connection between Association and AssociationEnd is {2, 3, 1, 1} which produces the partition pairs as follows:

[2][1, 1] and

[3][1,1,1].

The multiplicity constraint of the relationship Name between AssociationEnd and String is {1, 1, 0, 1}. This produces the partition pairs as follows:

[1, 1, 1][3],

[1, 1, 1][2, 1] and

[1, 1, 1][1, 1, 1].

5.2 Graphicalisation

Graphicalisation is the term used to refer to the process of creating a graph when the in- and out-degree of each node in the graph is known, but the exact edges of the graph are unknown. We use this term to refer to the process of creating bipartite graphs from a partition pair which indicates the degrees of the nodes in the resulting bipartite graphs.

The problem of graphicalisation can be viewed as a special case of network flow problem [11] where a network N is a digraph with two extra nodes a source node and a sink node. In our setting, we have a bipartite graph of two sets of nodes that is, the source domain and the target domain. Each of the nodes in the source domain has a set of edges each of which has capacity one connecting to each and every node in the target domain. The source node also has a set of edges to every node in the source domain. The capacities of these edges are set according to the values in the first partition of the given partition pair. Each node in the target domain also has an edge to the sink node where the capacity of each edge is set according to the value of each part in the second partition of the given partition pair.

Consider further the example from well-formedness rule 2, the source domain AssociationEnd of model-elements {end0, end1, end2} and the target domain String of model-elements {$n0$, $n1$, $n2$}, given a partition pair [1, 1, 1][2, 1], we can depict this setting in Fig. 8.

Having formulated this problem into a special case of network flow, we use Fig. 9 to enumerating all bipartite graphs from a partition pair. The algorithm takes one source node at a time to pick the required target nodes from the target domain. Every time the algorithm tries all possible

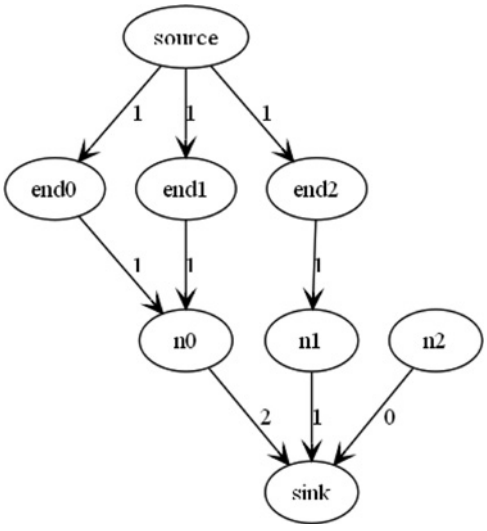

Fig. 8 *Network flow of a partition pair*

© The Institution of Engineering and Technology 2011

```
Input: A partition pair P of two integer partition S and T
Output: A set of all bipartite graphs that can be graphicalised using P
1:      create a queue Q
2:      let i = 0
3:      create a set C of all s_i–combinations of T
4:      for each combination c in C
5:              create a node n, let n.s=s_i; n.Ts=c; n.indegrees=T - d(c)
6:              enqueue n in Q
8:      endfor
9:      while Q.size > 0 and i<|S| do
10:             dequeue n from Q
11:             let last = n.next* (transitive closure, to the last node in n)
12:             let i = i++
13:             create a set C of all s_i–combinations of T
14:             for each combination c in C
16:             create a node n', let n'.s=s_i; n'.Ts=c; n'.indegrees= n.indegrees - d(c)
17:             let n.next = n'
18:             enqueue n in Q
19:     endwhile
```

Fig. 9 *Enumerating all bipartite graphs from a partition pair*

n-combination ways of picking different nodes. This continues until the last target node is selected. If a graph is invalid at any step in this process, it is immediately discarded.

5.3 Multiplication

The third step is to combine the bipartite graphs produced from each relationship to form the set of complete models. The operation union is employed to create the Cartesian product of the graphs generated in the graphicalisation step.

Definition 4: The union operation – a graph AG and a bipartite graph BG is defined as a new graph G having nodes $V(G) = V(\text{AG}) \cup V(\text{BG})$ and edges $E(G) = E(\text{AG}) \cup E(\text{BG})$. Suppose that, there are n different permutations $b_{1..n}(S(\text{BG}))$, each of which has its source vertices $S(\text{BG})$ assigned with a different labelling, we have the graph BG_i for a permutation $b_i(S(\text{BG}))$. To generate all graphs of AG ∪ BG, the union of the graph AG and the bipartite graph BG is defined as a set of graphs Gs where

$$Gs = \text{AG} \cup \text{BG} = (\text{AG} \cup b_0(S(\text{BG})), \text{AG} \cup b_1(S(\text{BG})), \ldots, \text{AG} \cup b_n(S(\text{BG})))$$

In our running example, let a graph AG in Fig. 10 be a graph between Association and AssociationEnd and a graph BG in Fig. 10 be a bipartite graph between AssociationEnd and String, the union of AG and BG must then consist of six models shown in Fig. 11. These models can be classified into only two equivalence classes A and B each of which consists of isomorphic members – class A consists of the model 10a and 10c and class B the model 10b, 10d, 10e and 10f.

Fig. 12 describes the partition–graphicalisation–multiplication method. There are two arguments on the correctness of this method. The first argument is the soundness of the test cases that is, no invalid test cases are generated; in other word, the method generates only the models appropriate to the UML metamodel. The second argument is that the test suite is complete that is, at least one model is generated from each equivalence class.

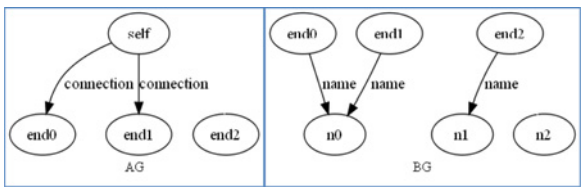

Fig. 10 *Two bipartite graphs for the union operation*

Soundness – line 3 calculates the range $m \ldots n$ of the possible numbers of edges considering the numbers of model elements and the given multiplicity constraints. Line 5 and 6 creates partition pairs and assigns in- and out-degrees of each node with respect to the range $m \ldots n$ and the multiplicity constraint of each node. In line 9, a set of bipartite graphs graphicalised from each partition pair using Fig. 9 is constrained by the degrees of each node. Because all test cases are constructed by joining these bipartite graphs together, it follows that the test cases are also valid.

Completeness – to show that the test suite produced is complete, there are three observations on the partition–graphicalisation–multiplication method. Firstly, the set of all partition pairs created from each relationship is complete. All partitions of the source domain and of the target domains are combined as a set of Cartesian products in line 7 to create a complete set of possible partition pairs. Secondly, all possible bipartite graphs are generated from a partition pair using Fig. 9. Therefore the bipartite graphs graphicalised are guaranteed to cover all possible configurations. Thirdly, the union operation in line 11 ensures that all configurations arise from combining two graphs – the front graph and the concatenating graph. By permutating all the source nodes of the concatenating graphs, all possible configurations that can be constructed by joining two graphs are created. Some of these graphs can be discarded when they are found to be an isomorphism of another graph. Fortunately, checking for isomorphism in the union operation can be done effectively since only a small number of nodes in the concatenating graph are needed to be considered.

6 Test classification and concretisation

JULE uses Crocopat to classify the output from the partition–graphicalisation–multiplication method into demonstrations and counterexamples. Beyer *et al.* [12] describes Crocopat as a high-level BDD package that allows querying and manipulating of graphs and other relational structures through RML, the language based on first-order predicate calculus. JULE translates an OCL well-formedness rule into an RML program. In our running example, the well-formedness rule 'self.allConnections()- > forAll($r1$, $r2$/ $r1$.name = $r2$.name implies $r1$ = $r2$)' would be translated into an RML program in Fig. 13 below.

JULE then uses each output from the partition–graphicalisation–multiplication method to create its representation in Rigi standard format (RSF) [13]. For example, the model 10a in Fig. 11 can be represented in RSF as shown Fig. 14.

CrocoPat interprets the RML program in Fig. 13 to create a BDD which consists of a set of variable nodes corresponding to the model elements and their properties to which the well-formedness rule refers, together with two terminal nodes labelled 0 and 1. Crocopat then reads the RSF files and evaluates them using the BDD created. According to the evaluation reached at one of the terminal node, each of the test models is marked as to whether it is a demonstration or a counterexample. For instance, the model 10a is marked as a counterexample.

Then, each pair of the test model and its expected test result (demonstration or counterexample) is concretised as a JUnit

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2010.0032

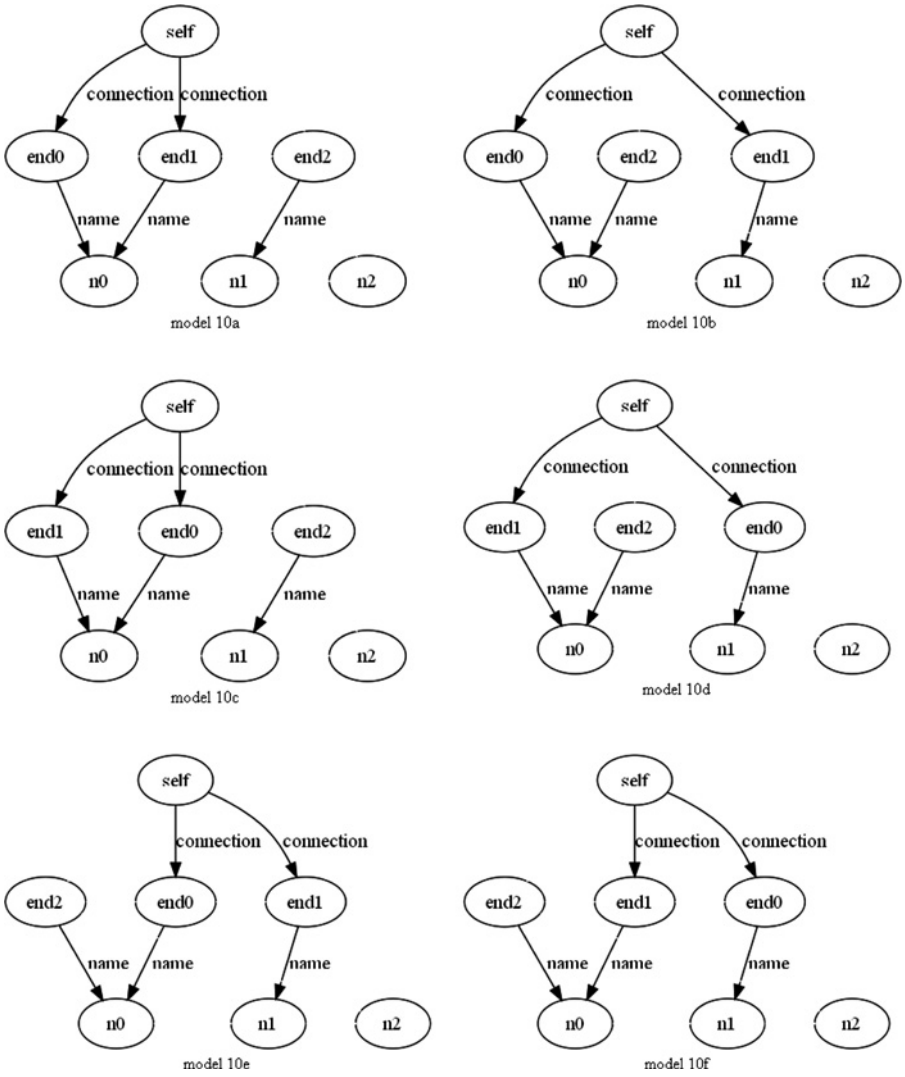

Fig. 11 *Union of graph AG and BG in Fig. 10 consists of six models*

test case. The JUnit generator uses Velocity [14] as its code generating engine and replaces the variables in the Velocity template with the object names and types, the names of the methods for creating links between the objects, and the expected test result for the test case.

An example of the JUnit test case is shown previously in Fig. 7. Each test case starts with the code that creates the test model in the tool's repository, follows by the code that invokes the model validation function of the tool, and finishes with the assertion code that checks if the validation result corresponds with the expected test result.

```
Input a set R of relationships r(S,T,s,t,l,u,v,w)
      let S be the domain of the source of r, T the domain of the target of r
      let s be the cardinality of S, t the cardinality of T,
      l be the lower bound of the multiplicity of r, u the upper bound,
      v be the lower bound of the multiplicity of r in the opposite direction
      i.e. the relationship from T to S  and w the upper bound
Output a list L of digraphs
1:    for  each relationship r
2:    find the range m..n of the possible numbers of edges under the multiplicity constraint
3:    where m = (s * v > t * l) ? s * v : t * l and n = (s * w > t * u) ? t * u : s * w
4:          for each possible number of links m .. n
5:                create a set of integer partitions Ps from (s, t, l, u, v, w)
6:                create a set of integer partitions Ps from (t, s, v, w, l, u)
7:                create a set of partition pair Ps × Pt
8:                      for each partition pair p in Ps × Pt
9:                            create a set of bipartite graphs BG
10:                           if L is empty add BG to L
11:                           else union each bg in BG with all existing digraphs in L
12:   return L as the list of all digraphs generated
```

Fig. 12 *Partition–graphicalisation–multiplication method*

Using the core package of the UML 1.4 specification, JULE generates 15 test suites, each of which is generated for a particular well-formedness rule. In total, there are more than 3000 test cases. Executing these test cases in ArgoUML, some previously unknown non-compliance issues are uncovered. We report our experiment and these non-compliance issues in the next section where the test results are analysed and the diagnosis of the causes of non-compliance is discussed.

7 Testing ArgoUML

ArgoUML [15] is a major open-source tool that supports the UML 1.4 standards specification, provides model analysis feature and is available under the Berkeley software distribution (BSD) licence. 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 [16] which

```
nameOk(s) := Association(s) &
                  FA( r1,r2,n, (connection(s,r1) & connection(s,r2) &
                        r1=r2) ->  name(r1,n) & name(r2,n) );
PRINT nameOk(s);
```

Fig. 13 *RML program for the well-formedness rule 2*

© The Institution of Engineering and Technology 2011

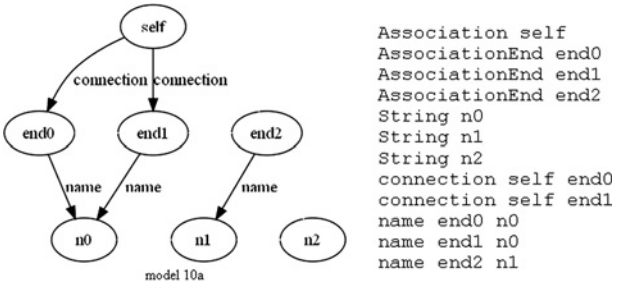

Fig. 14 *RSF representation of a test model*

directly supports MOF and uses the machine readable version of the UML 1.4 specification provided by the OMG'.

ArgoUML employs two methods for analysing design models, first the design critics which analyse 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. 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 analysed.

7.1 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 in Fig. 15.

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 two failures that were both demonstrations. One of them was shown in Fig. 16 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 overconstrained.

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

Fig. 15 *Well-formedness rule for AssociationEnd*

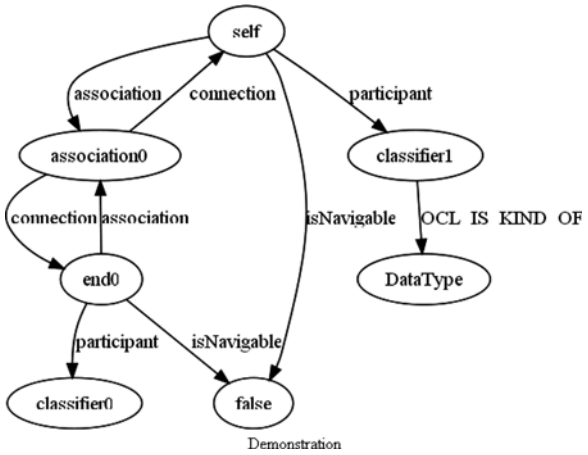

Fig. 16 *Test case for the AssociationEnd well-formedness rule*

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 Fig. 17. The test results from the larger test suites were consistent with those of the smaller ones.

7.2 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 Fig. 18. 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 below – line 1 and 2 shows the erroneous conditions. The IllegalArgumentException was thrown from line 3–8.

7.3 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 Fig. 19.

For this rule, JULE generated only nine test cases from one AssociationEnd, three AggregationKinds – Aggregate, Composite and None and three integers: 0, 1 and 2. Because these integers 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 nine test cases reported two failures shown in Fig. 20. The two tests are the association ends that are composite and have upperbound zero and two, respectively. Clearly, both test cases are counterexamples; however, they went undetected.

7.4 Diagnosis II

Running these two test cases in Eclipse's debug mode found a problem in line 3 of code in Fig. 21 that always returns false no matter what the value of the variable multi is. Tracing to the getMaxUpper method discovered a fault – this method

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2010.0032

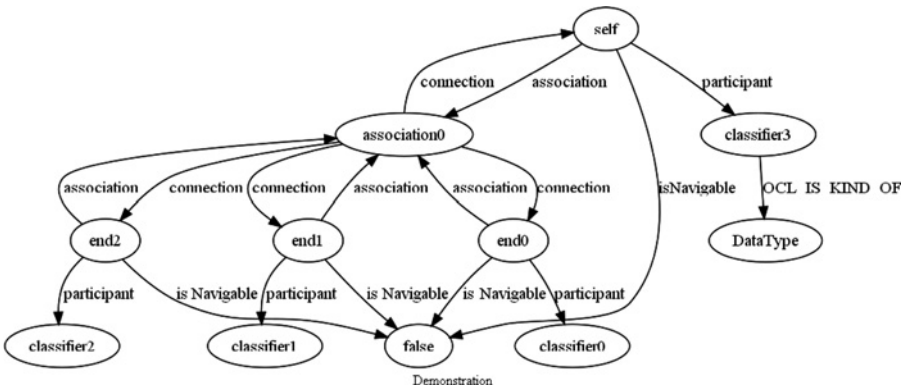

Fig. 17 *Larger test case for the AssociationEnd well-formedness rule*

```
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  }
```

Fig. 18 *Code snippet from the buildAssociationEnd method*

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

Fig. 19 *Another well-formedness rule for AssociationEnd*

always returns 0. This can be fixed easily by changing line 9 of the code in Fig. 22 to return max and ArgoUML can detect all counterexamples correctly.

7.5 Non-complaince Issue III

The next issue was one of the rules that constrain the semantics of Generalisation. This rule simply states that 'Circular inheritance is not allowed'. The OCL of this well-formedness rule is shown in Fig. 23. This rule excludes the self-element from being in one of its allParents.

The four test cases shown in Fig. 24 are counterexamples where self was involved, at some point, in a circular inheritance. In the first model in Fig. 24*a*, self is a child of itself. In the model in Fig. 24*b*, self has a parent that is a child of itself through a generalisable element. In Figs. 24*c* and *d*, self is a grandparent and great-grandparent of itself. All these models are invalid; however, ArgoUML can detect the cases of circular inheritance in Fig. 24*b* only.

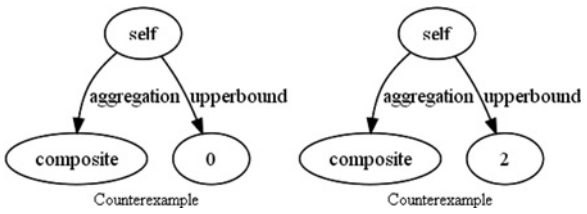

Fig. 20 *Two test cases for AssociationEnd*

```
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 }
```

Fig. 21 *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 }
```

Fig. 22 *getMaxUpper method*

```
not self.allParents->includes(self)
```

Fig. 23 *Well-formedness rule for GeneralisableElement*

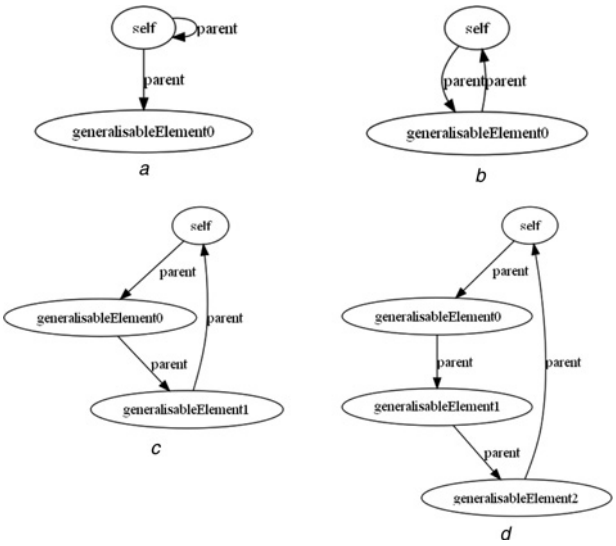

Fig. 24 *Test cases for GeneralisableElement*
a–d Counterexample

© The Institution of Engineering and Technology 2011

7.6 Diagnosis III

The buildGeneralisation method is shown in Fig. 25. 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)'.

Next, consider the cases in Figs. 24*c* and *d*, the grandchild and great-grandchild circular inheritances. The code that handled these non-compliance issues was implemented in another part of the buildGeneralisation method as shown in Fig. 26.

Generalisation gen : parent.getGeneralisation()

takes all generalisations 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 generalisable elements inherited by this generalisable element (the transitive closure), excluding the GeneralisableElement itself'. The implementation in the buildGeneralisation method deviates from this statement; this implementation only expresses the OCL in Fig. 27, but not equivalent to the original statement.

It was pointed out by a member of the ArgoUML team that circular Generalisation could be handled by one of the critics instead of by the build method. We tested ArgoUML with the model in Figs. 24*c* and *d* and found that there is a critic reporting problems in these models. Using this critic, ArgoUML can deal with all four cases of circular inheritance correctly. It can then be concluded that ArgoUML is compliant with this well-formedness rule.

8 Related works

8.1 Bounded exhaustive-testing and other techniques

Our compliance testing approach was inspired by a trend in software testing developed for programs that take complex data structures as input. Tools such as TestEra [17] and Korat [18] generate test cases using pre-conditions of program specifications to generate test data and post-conditions as test oracles.

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

Fig. 25 *Code snippet from the buildGeneralisation method*

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

Fig. 26 *Another part of code snippet from buildGeneralisation*

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

Fig. 27 *Deviation of the GeneralisableElement well-formedness rule*

TestEra uses Alloy Analyzer [19] to enumerate test cases and uses symmetry breaking predicates to generate only one non-isomorphic test case from each equivalence class of the test data that has the same structure. Korat also generates the test data effectively by monitoring access to all the fields of the candidate input and pruning the search tree to avoid paths that lead to isomorphism. The test generation of Korat differs from the one in this work. Test cases in Korat are checked for their isomorphism by comparing the objects' identities. The partition–graphicalisation–multiplication algorithm does not take objects' identities into consideration; it uses the degrees of nodes to produce non-isomorphic graphs.

In [20], TestEra was used for testing the Galileo tool [21] as an experiment on the practicality of bounded-exhaustive testing. This study shows a positive result that meaningful test cases, the fault trees – in this case, can be generated when the scope of the input is large enough.

Close to bounded-exhaustive testing is the test generation by disjunctive normal form (DNF) partitioning. Aichernig *et al.* [22] takes a subset of OCL and provides a method for partitioning OCL statements, then uses a constraint solver to generate test cases for mutation-testing according to the DNF. Another line of research from Farchi *et al.* [23] demonstrates test suite generation for parts of the POSIX standard and for the Java exception handling specification. Their method derives behavioural models from standards specifications. In contrast, JULE focuses on the static semantics part of modelling language specifications.

8.2 Lightweight formal method and other analysis tools

In the lightweight formal method [24], formal models are checked for consistency and correctness by way of testing based on input generated within only a small scope. A number of tools and techniques have been developed to support lightweight software methodology including the Alloy Analyser and the USE tool [25, 26] for UML. Analysis problems of UML models are one of the major research issues in the area of automated software engineering. A large body of research has been developed in the past few years.

One standard approach is to transform a UML model into a model in a semantic domain that is formally defined and supported by verification tools already. For example, in [27], UML state charts are mapped to communicating sequential processes (CSP). Consistency constraints of the UML state charts can then be specified and validated using the language and tools for CSP. Baresi and Pezzè [28] demonstrate the translation of object-oriented models to Petri nets that enable consistency checking and verification using tools for state exploration and concurrency analysis. In [29], UML interaction diagrams are transformed into automata for model checking to verify whether their interactions can be satisfied. The translation is implemented in the UML model checking tool HUGO/RT that is supported with SPIN [30].

In [31], UML models are represented in Z. Using the Z/Eves theorem prover [32], these models can be analysed for consistency of class properties by way of proving their initialisation and their states by calculating pre/post conditions of their operations. The Java modelling language (JML) is another widely used technology that enables this approach. Hamie [33] presents a mapping of OCL types, operations and collections to an implementation of JML library. This allows the translation of OCL constraints to JML constraints. This JML specification can be directly annotated in a Java program in the form of invariants and

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2010.0032

pre/post conditions. These annotations are then translated for checking the behaviour of the program at runtime that is, runtime assertion [34]. A number of tools that support JML assertions are listed in [35].

There are also other approaches for validating UML models that do not depend on existing verification tools. Hölscher *et al.* [36] present an approach for translating a given UML/OCL model into a graph transformation system to allow the UML models to be executed by applying graph transformation rules on the derived graph. With this approach, it is possible to validate the models by simulating the execution step-by-step.

8.3 Formal semantics of UML and OCL

The progress in automated testing based on the UML standards specification was hindered, to some extent, by the fact that the UML standards specification was not very precise and the OCL specification is incorrect at places. Much work has been done on the formalisation of UML such as [37, 38]. In UML 2.0, package merge is defined as a relationship between two packages by which the contents of the merged package is subsumed into the contents of the merging package. In [39], package merge was formalised using Alloy and analysed with the Alloy Analyser. This reveals falsity in the semantics of package merging. In particular, it was shown that the commutativity between the merged and merging package, in other word merge(x, y) = merge(y, x) for all x and y does not hold.

In term of dynamic models, [40] points out the differences among the three variations of state chart diagrams that is, UML, classical and rhapsody state charts. The key syntactic and semantic differences were clarified to demonstrate that a well-formed model in one variation may be considered ill formed in another. Thus, well-formed models may also be interpreted differently in different formalisms.

9 Lesson learned, conclusion and future works

It is shown in this work that black-box, bounded exhaustive-testing using both demonstrations and counterexamples is a sound approach for compliance assessment of software modelling tools. 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 [41] where a proof is constructed on a case-by-case basis until all required cases are proved.

While increasing the bounds on the input size to reach higher measure of coverage, the size of a test suite can grow unmanageably. Our technique generates only non-isomorphic test suites that provide the same coverage but are significantly reduced in size. Complexity analysis of our technique is being performed to demonstrate the effect of test suite reduction.

In this paper, we also set out to test the feasibility of using our technique to a realistic software tool. The basis of this evaluation was an experiment on applying a test suite generated from JULE to the ArgoUML modelling tool. The results reveal three previously unknown faults in ArgoUML. The first issue was corrected by the ArgoUML team and removed from the source code revision 16 250. The remaining issues were corrected in revision 16 693.

As a general observation, we note that the approach of translating these well-formedness rules to Java code manually 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 different model interpretations 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 [42, 43].

Our immediate future work is to experiment this framework with four UML 2.x tools including the tools Eclipse Galileo UML2 [44], Enterprise Architect 7.5 [45], Poseidon for UML 8.0 [46] and MagicDraw 16.6 [47]. We report our initial findings here as shown in Tables 2 and 3 to highlight some of the practical issues of standards compliance assessment – most current UML modelling tools do not sufficiently support the UML specifications and lack the required features to be tested for compliance.

Table 2 Comparison of four UML modelling tools: abstract syntax

Abstract syntax	Eclipse Galileo UML2	Enterprise Architect 7.5	Poseidon for UML CE 8.0	MagicDraw 16.6 SP2
multidirection of association	not support	support	not support	not support
generalisation of associations	support	support	not support	support

Table 3 Comparison of four UML modelling tools: well-formedness rules

Well-formedness rules	Eclipse Galileo UML2	Enterprise Architect 7.5	Poseidon for UML CE 8.0	MagicDraw 16.6 SP2
[1] an association specialising another association has the same number of ends as the other association	not applicable	not support	not applicable	not applicable
[2] when an association specialises another association, every end of the specific association corresponds to an end of the general association, and the specific end reaches the same type or a subtype of the more general end	not support	not support	not support	not support
[3] endType is derived from the types of the member ends	not support	not support	not support	not support
[4] only binary associations can be aggregations	not applicable	not support	not applicable	not applicable
[5] association ends of associations with more than two ends must be owned by the association	not applicable	not support	not applicable	not applicable

© The Institution of Engineering and Technology 2011

In Table 2, we assess two important features that are prerequisite for the well-formedness rules of Association. The first feature, the number of memberEnds allowed, is the ability to support multidirectional associations. Surprisingly, most tools, except Enterprise Architect, allow only bidirectional associations. In the second feature – generalisation of associations – lets an association to be generalised or specialised by another association. All tools except for Poseidon for UML allow generalisation of associations.

We then evaluate each tool for its modelling analysis features. Consider Table 3, the first column shows the five well-formedness rules of association. For each rule, there are three possible feature statuses – support, not support and not applicable – for example, rule 1 in the first row cannot be fully tested for compliance if the tool allows only bidirectional associations, thus marked as 'not applicable'. Enterprise Architect, the only tool that permits multidirectional association, does not support this rule. None of the tools support rule 2 and rule 3. Because Eclipse Galileo UML2, Poseidon for UML and MagicDraw only support bidirectional association, rule 4 and rule 5 do not apply to them. Neither of the rules is supported by Enterprise Architect.

Despite these initial findings, we intend to perform a thorough assessment of the four tools. We believe industry will continue to develop more advanced UML modelling tools and slowly integrate model analysis features into them. This also opens several lines of research such as the design of architectures for integrating model analysis components into existing software tools.

10 Acknowledgments

The authors would like to thank members of the ArgoUML team for constructive feedback in the ArgoUML case study, James Skene for his contributions to the development of JULE, Andrew Dingwall-Smith and Andy Maule for their helpful suggestions. We also thank the tool vendors for providing their tools under the evaluation licence agreements.

11 References

1 Eichelberger, H., Eldogan, Y., Schmid, K.: 'A comprehensive survey of UML compliance'. Lecture Notes in Informatics, Gesellschaft für Informatik, pp. 39–50
2 Object Management Group (OMG): 'The metaobject facility specification version 1.4.1', 2005
3 Object Management Group (OMG): 'UML 2.0 object constraint language (OCL) specification', 2003
4 SAE International: 'AS5506 architecture analysis & design language (AADL)'. Available at http://www.aadl.info/aadl, accessed 2009
5 Bunyakiati, P., Finkelstein, A., Skene, J., Chapman, C.: 'Using JULE to generate a compliance test suite for the uml standard'. Proc. Int. Conf. on Software Engineering 2008, Leipzig, Germany, 2008, pp. 827–830
6 Bunyakiati, P., Finkelstein, A.: 'The compliance testing of software tools with respect to the UML standards specification – the ArgoUML case study'. ICSE Workshop on Automation of Software Test, AST '09, Vancouver, Canada, 2009, pp. 138–143
7 Object Management Group (OMG): 'The unified modelling language (UML) version 1.4 specification', 2001
8 Beyer, D.: 'Relational programming with crocopat'. Proc. Int. Conf. on Software Engineering 2006, Shanghai, China, 2002, pp. 807–810
9 Bryant, R.E.: 'Symbolic Boolean manipulation with ordered binary decision diagrams', *ACM Comput. Surv.*, 1992, **24**, (3), pp. 293–318
10 JUnit, available at http://www.junit.org/, accessed May 2010
11 Ahuja, R.K., Magnanti, T.L., Orlin, J.: 'Network flows: theory, algorithms, and applications' (Prentice Hall, 1993)
12 Beyer, D., Noack, A., Lewerentz, C.: 'Efficient relational calculation for software analysis', *IEEE Trans. Softw. Eng.*, 2005, **31**, (2), pp. 137–149
13 Wong, K.: 'The Rigi user's manual – version 5.4.4'. Available at http://www.rigi.cs.uvic.ca/downloads/rigi/doc/user.html, accessed 1998
14 Apache Software Foundation: 'The Apache velocity project'. Available at http://velocity.apache.org/, accessed May 2010
15 ArgoUML, available at http://argouml.tigris.org/, accessed September 2010
16 Sun Microsystems: 'The JavaTM metadata interface (JMI) specification', 2002
17 Khurshid, S., Marinov, D.: 'TestEra: specification-based testing of Java programs using SAT', *Autom. Softw. Eng.*, 2004, **11**, (4), pp. 403–434
18 Boyapati, C., Khurshid, S., Marinov, D.: 'Korat: automated testing based on Java predicates'. Proc. 2002 ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA'02), New York, NY, USA, 2002, pp. 123–133
19 Jackson, D.: 'Software abstractions: logic, language, and analysis' (MIT Press, 2006)
20 Sullivan, K., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: 'Software assurance by bounded exhaustive testing', *IEEE Trans. Softw. Eng.*, 2005, **31**, (4), pp. 133–142
21 Coppit, D., Sullivan, K.J.: 'Galileo: a tool built from mass-market applications'. Proc. 22nd ICSE, Limerick, Ireland, 2000, pp. 750–753
22 Aichernig, B.K., Salas, P.A.P.: 'Test case generation by OCL mutation and constraint solving'. Proc. Int. Conf. on Quality Software (QSIC), Melbourne, Australia, 2005, pp. 64–71
23 Farchi, E., Hartman, A., Pinter, S.S.: 'Using a model-based test generator to test for standard conformance', *IBM Syst. J.*, 2002, **41**, (1), pp. 89–110
24 Agerholm, S., Larsen, P.G.: 'A lightweight approach to formal methods'. Proc. Int. Workshop on Current Trends in Applied Formal Methods, Germany, 1998, pp. 168–183
25 Gogolla, M., Bohling, J., Richters, M.: 'Validation of UML and OCL models by automatic snapshot generation'. (Springer, 2003), (*LNCS*, **2863**), pp. 265–279
26 Gogolla, M., Büttner, F., Richters, M.: 'USE: a UML-based specification environment for validating UML and OCL', *Sci. Comput. Program*, 2007, **69**, (1–3), pp. 27–34
27 Engels, G., Heckel, R., Küster, J.M.: 'Rule-based specification of behavioral consistency based on the UML meta-model'. UML 2001, (*LNCS*, **2185**), pp. 272–286
28 Baresi, L., Pezzè, M.: 'Improving UML with Petri nets', *Electron. Notes Theor. Comput. Sci.*, 2001, **44**, (4), pp. 107–119
29 Knapp, A., Wuttke, J.: 'Model checking of UML 2.0 interactions'. Proc. 2003 Int. Conf. on Models in Software Engineering, Berlin, Heidelberg, 2007, pp. 42–51
30 Holzmann, G.J.: 'The SPIN model checker: primer and reference manual' (Addison-Wesley, 2003)
31 Amálio, N., Stepney, S., Polack, F.: 'Formal proof from UML models'. ICFEM 2004, (*LNCS*, **3308**), pp. 418–433
32 Saaltink, M.: 'The Z/EVES system', *Lect. Notes Comput. Sci.*, 1997, **1212**, pp. 72–85
33 Hamie, A.: 'On the relationship between the object constraint language (OCL) and the Java modeling language (JML)'. Seventh Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06), IEEE Computer Society, 2006, pp. 411–414
34 Cheon, Y., Leavens, G.T.: 'A runtime assertion checker for the Java modeling language (JML)'. Proc. Int. Conf. on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, 2002, pp. 322–328
35 Burdy, L., Cheon, Y., Cok, D.R., *et al.*: 'An overview of JML tools and applications', *Int. J. Softw. Tools Technol. Transf.*, 2005, **7**, (3), pp. 212–232
36 Hölscher, K., Ziemann, P., Gogolla, M.: 'On translating UML models into graph transformation systems', *J. Vis. Lang. Comput.*, 2006, **17**, (1), pp. 78–105
37 Clark, T., Evans, A., Kent, S.: 'The metamodelling language calculus: foundation semantics for UML'. Proc. Fourth Int. Conf. on Fundamental Approaches To Software Engineering, 2–6 April 2001, (*LNCS*, **2029**), pp. 17–31
38 Amelunxen, C., Schurr, A.: 'Formalising model transformation rules for UML/MOF 2', *IET Softw.*, 2008, **2**, (3), pp. 204–222
39 Zito, A., Diskin, Z., Dingel, J.: 'Package merge in UML 2: practice vs. theory?' in Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (Eds.): Model Driven Engineering Languages and Systems, (Springer, 2006), (*LNCS*, **4199**), pp. 185–199
40 Crane, M., Dingel, J.: 'UML vs. classical vs. rhapsody statecharts: not all models are created equal', *Softw. Syst. Model.*, 2007, **6**, (4), pp. 415–435
41 Young, M., Pezze, M.: 'Software testing and analysis: process, principles and techniques' (Wiley, 2007), p. 20

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2010.0032

42 Skene, J., Emmerich, W.: 'Specifications, not meta-models'. Proc. ICSE Workshop on Global Integrated Model Management, ACM/IEEE, Shanghai, China, 2006, pp. 47–54
43 Skene, J.: 'The UCL MDA Tool'. Available at http://uclmda.sourceforge.net/index.html, accessed September 2010
44 The Eclipse Project: 'The eclipse modelling framework (EMF)'. Available at http://www.eclipse.org/emf/, accessed 2009
45 Sparx systems: 'Enterprise Architect 7.5'. Available at http://www.sparxsystems.com/products/ea/7.5/, accessed June 2010
46 Gentleware, A.B.: 'Poseidon UML editor'. Available at http://www.gentleware.com/, accessed May 2010
47 No Magic, Inc.: 'MagicDraw 16.6'. Available at http://www.magicdraw.com/, accessed June 2010

© The Institution of Engineering and Technology 2011