Towards Validation of UML Design Model

A. Mishra and D. Yadav

Department of Computer Science and Engineering, MNNIT Allahabad, Allahabad
{ucashish1@gmail.com;dky@mnnit.ac.in}

Abstract. Design models of model-driven engineering support efficient and error free designing i.e. independent of implementation details. Ensuring design model consistencies is an important activity in software development. One way of achieving it, is by preventing introduction of inconsistencies in the initial phases of software development. This can be realized through checking of inconsistencies in model represented through UML class diagram. Design models are based on certain design rules. In our case validation of design model is done against these design rules to find inconsistencies in the model. Validation of design rules is based on predicate logic while validation of design models against validated design rules is based on XML (Extensible Markup Language) and XSD (XML Schema Definition). The method has been illustrated with a case study.

Keywords: UML, Design Rules, Design Model, Predicate Logic, Consistency.

1 Introduction

MDE (Model Driven Engineering) has been adopted on a large scale in the industrial projects. In MDE approach, firstly the design model of the software is developed. UML class diagram is an important means for representing the design model. There are some restrictions or constraints on these design models. These constraints are modeled in UML in the form of design rules. Inconsistent design rules may result in to inconsistent design models which in turn may lead to generation of software with errors.

A major concern to deal with these inconsistencies is to find the cause with accurate visualization. It might be possible that few or all model elements are responsible for cause of errors. As design models are based on design rules, hence both the design models and design rules must be validated to find inconsistencies. A better understanding of the cause may help to concentrate on the parts of the models which are responsible for the errors. To find inconsistencies in software, validation of design model is done which in turn requires validation of design rules.

This paper extends our previous work [1] which focuses on validation of design rules and design models both by using predicate logic approach. In this article XML approach has been used for the validation of design model. XML approach is easier to realize as compared to logic. To validate design rules that are in OCL (Object Constraint Language), first OCL expressions are converted in to predicate logic expression. These logical expressions are validated through tableaux method of predicate logic. Predicate logic has been used because of its capability to capture models accurately. For validation of design model, first a text file is prepared to represent UML class diagram. Further, this text file is used to prepare XML and XSD of the class diagram. Validation of XML file is done with respect to corresponding XSD file. Validating UML design model against valid design rules related to recursive property is the main goal of the paper.

B. Iyer, S. Nalbalwar and R. Pawade (Eds.)
ICCASP/ICMMD-2016. Advances in Intelligent Systems Research.
Vol. 137, Pp. 774-785.

2 Basic Concepts

This section provides the definitions of the concepts used in the paper. Subsection 2.1 describes the concepts related to UML class diagrams and OCL (Object Constraint Language). In subsection 2.2, the basic concepts and notations of logic used in the paper is summarized. A small description of tableaux method is provided in subsection 2.3. Subsection 2.4 presents the concepts related to XML and XSD.

2.1 Basic Concepts on UML and OCL

UML is a general-purpose modeling language to visualize, specify, construct and document software systems. It is not a programming language, however, there are different tools available that can be used to generate code in various languages using UML diagrams. In UML, a class diagram represents the static aspects of an application. Class diagram is not only used for visualizing, describing and documenting different aspects of a system but also for construction of executable code of the software applications. The class diagram describes the attributes, operations of a class and the constraints imposed on the system. Constraints which cannot be represented graphically in the class diagram, they may be expressed in OCL.

UML class diagram is not sufficient for an exact and clear specifications of the design model. There is need of additional constraints for the object in the model. If natural language is used to describe such constraints, this may lead to several ambiguities. To avoid ambiguities generally formal languages are used which is generally considered to be difficult. This gap is fulfilled by OCL.

The OCL is a standard query language, that is part of the UML set by the Object Management Group (OMG) as a standard for object oriented analysis and design. OCL is a formal language which is used to express side effect free constraints. In UML, OCL can be used to specify constraints and other expressions attached to the design models. The reason for selecting the OCL over other languages is, its simplicity to express constraints semi-formally.

2.2 Basic Concepts of Logic

This section provides a brief description about basic concepts and notations related to logic used in the paper. In computer science the goal of logic is to develop a language that is used to model situations that one faces as computer science professionals. Arguments can be constructed by reasoning about situations. Logic is about the validity of arguments and consistency among statements. In the abstract sense logic deals with the notion of truth.

Throughout the paper upper case lettersX, Y, Z...are used as variables. The lower case letters a,b,c ...denote constants. The collection of variables aredenoted by ;;; ...and ;;;...denote collection of constants. The characters p,q,r, ...denote predicate symbols. A term may be either a constant or avariable. An atom is represented by $p(T_1..., T_n)$ or $p(\text{-})$where n is known from the context, and p is a n-ary predicate and $T_1, ...,T_n$ are terms. An atom will be a ground atom if every T_i is a constant. An ordinary literal can be describedas an atom or a negated atom ¬ pr( ). The form $A_1\ \omega\ A_2$, where A_1 and A_2 are terms and ω is either <, ≤, >, ≥, = or <>, denotes built-in-literal.

A normal clause is of the form $A \leftarrow L_1 \wedge ... \wedge L_m$ with $m \geq 0$, where each $L_i$is a literal (either ordinary or built-in) and A is an atom. All variables occurring in A as well as in each L_i, are assumed to be universally quantified over the complete formula. $L_1 \wedge ... \wedge L_m$ is the body and A is called as Head of clause. Based on its various forms, a normal clause can be a deductive rule, a fact, or a condition, that are defined as the following.

A formula of the (denial) form $\leftarrow L_1 \wedge ... \wedge L_m$with $m \geq 1$ is called a **condition**. A situation that should not hold.

A normal clause of the form $p(\text{-}) \leftarrow L_1 \wedge ... \wedge L_m$ with $m \geq 1$ is called **deductive rule**, where p is a derived predicate defined by the deductive rule. The predicate symbol p is defined as the set of all deductive rules that have p in their head.

A **fact** is a normal clause of the form $p(\text{-}) \leftarrow$ or $(p(a))$, where $p(\text{-})$is a ground atom [2].

2.3 Overview of Tableaux method

Arguments can be validated through logic. Truth table is a method for showing validity of arguments. The problem with validation by truth tables is that truth table gets very large rapidly. A statement which has two variables will require four rows for its validation in truth table, and a statement that involves three variables will require eight rows. Therefore, the number of rows of a truth table doubles every time a new variable is added for considering all possible combination of argument's values. In general, a statement with n variables has 2^n rows in truth table, which causes exponential explosion.

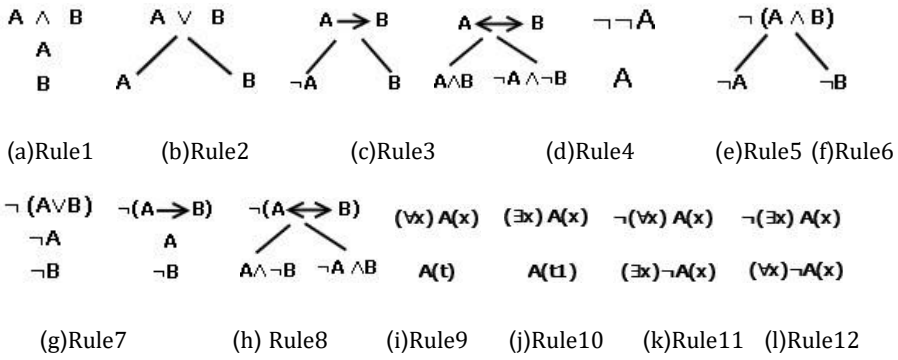
The tableaux method is based on the principle of negating the conclusion of an argument for checking its consistency with the premises. The basic idea is to check whether it can be possible for a false conclusion to be consistent with all premises being true. If it is not possible i.e. the conclusion must be true when the premises are true, it can be said that conclusion is semantically entailed by the premises. A semantic tableau is an organization of sequence of formulae in the form of a tree, that is constructed by following certain rules.

The rules are displayed in Fig. 1. where t is a term in Rule10 which is known as Universal Instantiation (UI). And t_1 is a term which has not been used in the derivation so far, in Rule11 that is known as Existential Instantiation (EI). Rule (14) When A and ¬A both appear in a branch of tableau, inconsistency is indicated and tableau cannot be further extended i.e. it is said to be closed. Statements in the original arguments are consistent, if the tableau is closed. An example showing the application of tableau method over the arguments is given below:

All men are mortal. Socrates is a man. Therefore, Socrates is mortal. Let M(x) stand for 'x is mortal' and (x) stand for 'x is a man' Then the argument can be symbolized as:

$(\forall)(H(x) \rightarrow M(x)) \wedge H(Socrates) \models M(Socrates)$

Adding the negation of conclusion with the premises and try for construction of a closed tableau as shown in Fig. 2 [3].



(a)Rule1 (b)Rule2 (c)Rule3 (d)Rule4 (e)Rule5 (f)Rule6

(g)Rule7 (h) Rule8 (i)Rule9 (j)Rule10 (k)Rule11 (l)Rule12

(m)Rule13

Fig.1. Rules for semantic tableau

2.4 Concept related to XML and XSD

XML is a W3C recommendations which stands for Extensible Markup Language. It is a content description language which describes data. In XML, one can define his own tags as per the requirements of the applications. XML tags describes the structure and semantics of data, not its formatting. It is used for exchanging information between various incompatible applications or database. DTD(Document type definition) or XML schema is used by XML to describe tags. An application program used to process the XML document is known as XML parser.

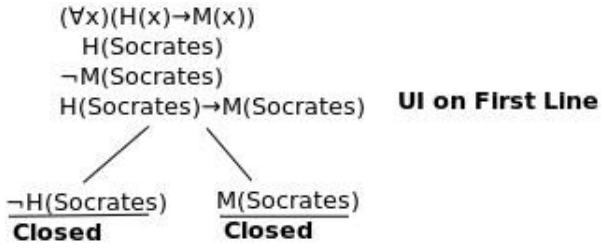


Fig. 2. A Simple Example

A schema is a formal description of what comprises a valid document. An XML Schema is an XML document containing a formal description of what comprises a valid XML document. The process of checking whether an XML document conforms to a schema is known as validation. XML schema is used to validate the content of XML document. XML parser uses following two techniques to process/validate the XML document.

First technique is DOM (Document Object Model) which represents XML document as a tree structure in which elements, text and attributes are represented as nodes. This technique is used, when we have sufficient amount of memory. The main advantage of this technique is that in DOM, back and forth movements are possible due to recursive property of tree.

Second technique is SAX (Simple API for XML) in which accessing of XML element is done sequentially. It is an event based parsing technique. This technique uses standard API to access the component of XML document. The disadvantage of it is that backward and forward movement is not possible. But latter technique requires less memory than former. In this paper we have used SAX approach based on java API for validation.

3 Validation of Design Rules

This section provides validation of some design rules given in [4]. For validation, the rules are first converted into predicate logic form following the approach of [2]. The approach given in [2] is summarized in subsection 3.1. Subsection 3.2 provides the list of two OCL design rules. Subsection 3.3 presents the detailed steps for converting these design rules into predicate logic. In Subsection 3.4, the validity of the design rules expressed in predicate logic form have been checked, by tableaux method. Finally, a case study is provided in Subsection 3.5.

3.1 Approach for Logic Formalization

We have translated design rules into predicate logic representation to validate them in OCL forms. Due to this above problem becomes equivalent to check the consistency of these predicates. The translation of OCL constraints into predicate logic involves two steps. First step is the transformation of each OCL expressions into an equivalent expression in terms of select and size. Select and size OCL operations are applied to set of elements, size returns the number of elements in the set and select returns the subset of the collection which is satisfied by the condition. Table 1 shows the OCL operations and gives their equivalent simplified expressions.

Once simplified (if needed) the original invariant, which we have now, will have a limited set of patterns (in terms of select and size). In the following, the original expression is denoted by expr and its translation into logic by Tr(expr). We assume path=$obj_0.r_1....r_n$ is a path starting from obj of a class C, navigating through roles r_1 to r_n, where r_n is a role or an attribute, resulting into a set of objects or values.

(a) expr = path

$Tr(expr) = c(Obj_0) \wedge assoc_1(..., Obj_0, ..., Obj_1, ..., Obj_k.) \wedge ... \wedge assoc_n(..., Obj_{n-1}, ..., Obj_n, ..., Obj_l)$

where $assoc_j(...,Obj_{j-1},...,Obj_j,...,Obj_k)$ is the logic representation of the association of arity k, with or without an association class, between roles r_{j-1} and r_j. The predicate $assoc_n$ is binary if it represents an attribute.

(b) expr = path opComp value

where value can be either a navigation path, a constant or an iteration variable, and opComp is a comparison operator. Then,

$Tr(expr) = Tr(path)\ [\wedge Tr(value)] \wedge Obj_1 opComp\ Obj_2$

(c) expr = path[→ select(expr)]→size() opComp k

where the comparison operator opComp is either <,>,= or <> and k is an integer not less than zero. Then, Tr(expr) will be based on opComp:

(c.1) $Tr(obj_0 .r_1 ...r_n[\rightarrow select(expr)]\rightarrow size()<k) = Tr(obj_0) \wedge \neg cond(Obj_0, X,..., X_m)$

$cond(Obj_0, X, ..., X_m) \leftarrow Tr_1(path)[\wedge Tr_1(expr)] \wedge ... \wedge Tr_k(path)\ [\wedge Tr_k(expr)] \wedge Obj_i <> Obj_j$

(c.2) $Tr(obj_0.r_1...r_n[\rightarrow select(expr)]\rightarrow size()>k) = Tr_1(path)[\wedge Tr_1(expr)] \wedge ... \wedge Tr_{k+1}(path)[\wedge Tr_{k+1}(expr)] \wedge Obj_i <> Obj_j$

(c.3) $Tr(obj_0.r_1...r_n[\rightarrow select(expr)]\rightarrow size()=k) = Tr_1(path)[\wedge Tr_1(expr)] \wedge \ldots \wedge Tr_k(path)[\wedge Tr_k(expr)] \wedge \neg cond(Obj_0, X,...,X_m)$

$cond(Obj_0, X,..., X_m) \leftarrow Tr_1(path)[\wedge Tr_1(expr)] \wedge ... \wedge Tr_{k+1}(path)[\wedge Tr_{k+1}(expr)] \wedge Obj_i <> Obj_j$

(c.4) $Tr(obj_0.r_1...r_{n1}.r_n[\rightarrow select(expr)]\rightarrow size()<>k) = Tr(obj_0) \wedge \neg cond(Obj_0, X,..., X_m)$

$cond(Obj_0, X, ..., X_m) \leftarrow Tr_1(path)[\wedge Tr_1(expr)] \wedge ... \wedge Tr_k(path)\ [\wedge Tr_k(expr)] \wedge Obj_i <> Obj_j \wedge \neg oneMore(Obj_0, X, ...,X_m)$

$oneMore(Obj_0, X,...,X_m) \leftarrow Tr_1(path)[\wedge Tr_1(expr)] \wedge ... \wedge Tr_{k+1}(path)\ \wedge Tr_{k+1}(expr)\ ] \wedge Obj_i <> Obj_j$

where Obj_0 represents the object at the starting of the path, and Obj_i represents an object obtained as a result of the navigation path according to $Tr_i(path)$. Eachexpression Tr_irefers to the same translation but using different variables for all terms different from the initial object Obj_0. The expression $Obj_i <> Obj_j$, $1 \leq i < j \leq k$ (k+1in (c.2), (c.3)), and the second derivation rule of (c.4)) ensures that different variables do not represent the same object. The variables X, ...,X_m needed by each auxiliary (derived) predicate are the iteration variables of all the select operations in which the translated expr is included.

(d) expr = $expr_1$ and $expr_2$

Then, $Tr(expr) = Tr(expr_1) \wedge Tr(expr_2)$

(e) expr = $expr_1$ or $expr_2$

Then, Tr(expr)=disjunction(Obj_0, X, ..., X_m)

where disjunction is a derived predicate defined by the rules:

$disjunction(Obj_0, X, ..., X_m) \leftarrow Tr(expr_1)$

$disjunction(Obj_0, X,.., X_m) \leftarrow Tr(expr_2)$

(f) expr = not expr

In this case, Tr(expr) depends on the kind of expression:

(f.1) Tr(not path[[→select(expr)]→size()] opCompvalue)=

Tr(path[→select(expr)]→size() invOpComp k)

where invOpComp is the inverse of opComp, that is, if opComp is > then invOpComp is ≤, and so on, and value can be either a path, a constant, or an iteration variable.

(f.2) Tr(not expr)= Tr(exprdM)

where exprdM is the expression resulting from iteratively applying DeMorgan's law on not expr.

Table 1. Equivalences of OCL Operations

Original Expression	Equivalent Expression with select and size
path→includes(obj)	path→select(e|e=obj)→size()>0
path→excludes(obj)	path→select(e|e=obj)→size()=0
path→includesAll(c)	c→forall(e|path→includes(e))
path→excludesAll(c)	c→forall(e|path→excludes(e))
path→isEmpty()	path→size()=0
path→notEmpty()	path→size()>0
path→exist(e|body)	path→select(e|body)→size()>0
path→forall(e|body)	path→select(e|not body)→size()=0
path→isUnique(e|body)	path→select(e|path→select(e2|e<>e_2 AND e_2.body=e.body))→size()=0
path→one(e|body)	path→select(e|body)→size()=1
path→reject(e|body)	path→select(e|not body)

3.2 Design Rules

The two UML design rules with their OCL form taken from [4] are as below:
Design Rule 1: At most one Association End may be an Aggregation or Composition.
OCL Form:

context Association inv:
self.memberEnd→size()>0 implies self.memberEnd→
select(p|p.aggregation<>AggregationKind::none)→size()≤1

Design Rule 2: No circular Inheritance allowed.
OCL Form:

context Class inv : not self.allParents()→includes(self)

3.3 Logic Conversion of Design Rules

The detailed steps for conversion of the design rules into predicate logic is shown below:
Design Rule 1: Logic Conversion:

let expr=self.memberEnd→size()>0 implies self.memberEnd→ select(p|p.aggregation <>AggregationKind::none)→size()≤1

since p→q~=¬pVq, so

expr=¬(self.memberEnd→size()> 0)or(self.memberEnd→
select(p|p.aggregation<>AggregationKind::none)→size()≤1)

since p≤q⇒p<(q+1) and using (f.1)

expr=(self.memberEnd→size()≤0)or(self.memberEnd→
select(p|p.aggregation<>AggregationKind::none)→size()<2)

since p≤q⇒p<(q+1)

expr=(self.memberEnd→size()<1)or(self.memberEnd→
select(p|p.aggregation<>AggregationKind::none)→size()<2) expr=$expr_1$ or $expr_2$

where $expr_1$=self.memberEnd→size()<1
$expr_2$=self.memberEnd→select(p|p.aggregation<>AggregationKind::none)→size()<2
$expr_1$=self.memberEnd→size()<1

using (c.1)

Tr($expr_1$)=association(a)∧¬cond(a,me)
cond(a,me)←association(a)∧memberEndOf(a,me)$expr_2$=self.memberEnd→select(p|p.aggregation<>AggregationKind::none)→size()<2

using (c.1)

Tr($expr_2$)=association(a)∧¬$cond_1$(a,me,ag,agk)
$cond_1$(a,me,ag,agk)←association(a)∧memberEndOf(a,me)∧aggregationOf(me,ag)
∧AggregationKindOf(ag,agk)∧ag<>(agk=none)∧memberEndOf(a,me_1)∧
aggregationOf(me_1,ag_1)∧aggregationKindOf(ag_1,agk_1)∧ag_1<>(agk_1=none)∧
ag<>ag_1)∧(agk<>agk_1)∧(me<>me_1)
expr=$expr_1$ or $expr_2$

using (e)

Tr(expr)=disjunction(a, me, ag, agk)
disjunction(a,me,ag,agk)← Tr($expr_1$)
disjunction(a,me,ag,agk)← Tr($expr_2$)

Design Rule 2: Logic Conversion:

let expr=not self.allParents()→includes(self)
from Table 1
expr=not self.allParents()→select(e|e=self)>0
using (f.1)
expr=self.allParents()→select(e|e=self)≤0
since p≤q⇒p<(q+1)
expr=self.allParents()→select(e|e=self)<1
using (c.1)
Tr(expr)=class(c)∧¬cond(c, ap, e)
cond(c, ap, e)←class(c)∧allParentsOf(c, ap)∧elementOf(ap,e)∧(e=c)

3.4 Validation of Design Rules

Validation of the design rules by semantic tableaux method is illustrated below. Design rules can be validated by negating the conclusion and taking logical AND of it with the premises. If all the branches of the semantic tableau are closed, we can infer that negation of conclusion is inconsistent with the rules (premises).

Hence, the conclusion is consistent with rules.

Design Rule 1in logic form after replacing values of derived predicate:

Tr(expr)=disjunction(a,me,ag,agk))

disjunction(a,me,ag,agk)←association(a)∧(¬association(a)∨¬memberEndOf(a,me)
)
disjunction(a,me,ag,agk)←association(a)∧(¬association(a)∨¬memberEndOf(a,me)
∨¬aggregationOf(me,ag)∨¬aggregationKind(ag,agk)∨¬ag<>(agk=none)∨
¬memberEndOf(a,me_1)∨¬aggregationOf(me_1,ag_1)∨¬aggregationKindOf(ag_1,agk)
∨¬(ag_1<>(agk_1=none))∨¬(ag<>ag_1)∨¬(agk<>agk_1)∨¬(me<>me_1))

Invariant in design rule 1 is for association hence this rule must be true for all associations. So, it can be transformed into the following form: (∀a)(association(a)→disjunction(a,me,ag,agk))
disjunction(a,me,ag,agk)←association(a)∧(¬association(a)∨¬memberEndOf(a,me))
disjunction(a,me,ag,agk)←association(a)∧(¬association(a)∨¬memberEndOf(a,me)∨¬aggregationOf(me,ag)∨
¬aggregationKind(ag,agk)∨¬ag<>(agk=none)∨
¬memberEndOf(a,me_1) ∨¬aggregationOf(me_1,ag_1)∨¬aggregationKindOf(ag_1,agk_1)
∨¬(ag_1<>(agk_1=none))∨¬(ag<>ag_1)∨¬(agk<>agk_1)∨¬(me<>me_1))

Having as as an instance of association:

(∀a)(association(a)→disjunction(a,me,ag,agk))∧association(as) ⊨disjunction(as,me,ag,agk))
disjunction(as,me,ag,agk)←association(as)∧(¬association(as)∨¬memberEndOf(as,me))
disjunction(as,me,ag,agk)←association(as)∧(¬association(as)∨¬memberEndOf(as,me)∨¬aggregationOf(me,ag)∨¬ag
gregationKind(ag, agk)∨¬ag<>(agk=none)∨¬memberEndOf(as,me_1)∨¬aggregationOf(me_1,
ag_1)∨¬aggregationKindOf(ag_1,agk_1) ∨¬(ag_1<>(agk_1=none))∨¬(ag<>ag_1)∨¬(agk<>agk_1)∨¬(me<>me_1))

Added the negation of conclusion to the premises and constructed a closed tableau as shown in Fig. 3.

Design Rule 2in logic form after replacing values of derived predicate :

Tr(expr)=class(c)∧(¬class(c)∨¬allParentsOf(c,ap)∨¬elementOf(ap,e)∨¬(e=c))

Invariant in design rule 2 is for class hence this rule must be true for all classes. So, it can be transformed into the following form :

(∀c)(class(c)→class(c)∧(¬class(c)∨¬allParentsOf(c, ap)∨¬elementOf(ap, e)∨¬(e=c)))

Having A as an instance of a class :

(∀c)(class(c)→class(c)∧(¬class(c)∨¬allParentsOf(c, ap)∨¬elementOf(ap, e) ∨¬(e=c)))

∧class(A) ⊨(class(A)∧(¬class(A)∨¬allParentsOf(A,ap)∨¬elementOf(ap, e)∨¬(e=A)) Added the negation of conclusion to the premises and constructed a closed tableau as shown in Fig. 4.

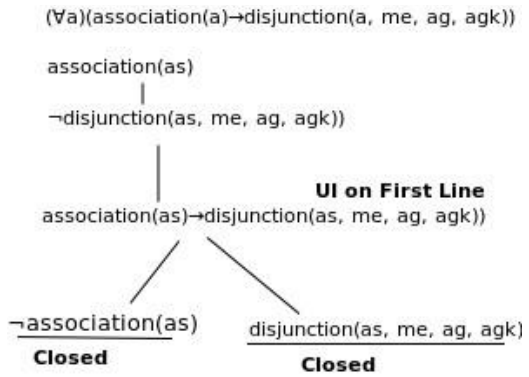


Fig. 3. Closed tableau of Design Rule 1 with negation of conclusion

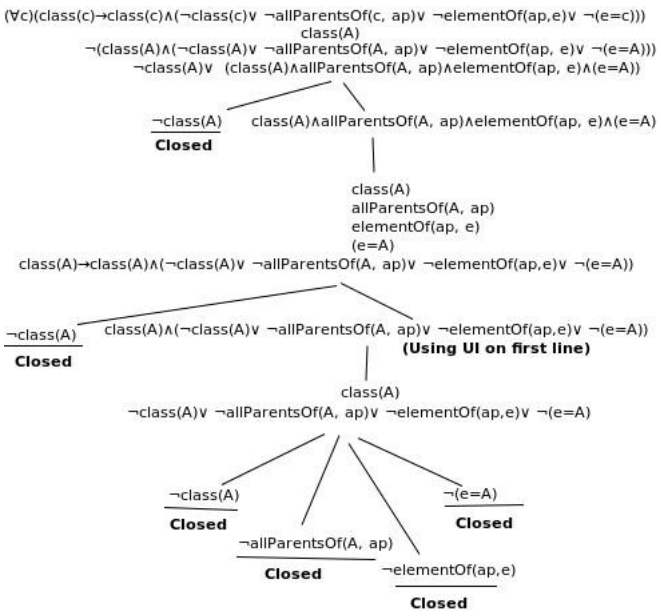


Fig.4. Closed tableau of Design Rule 2 with negation of conclusion

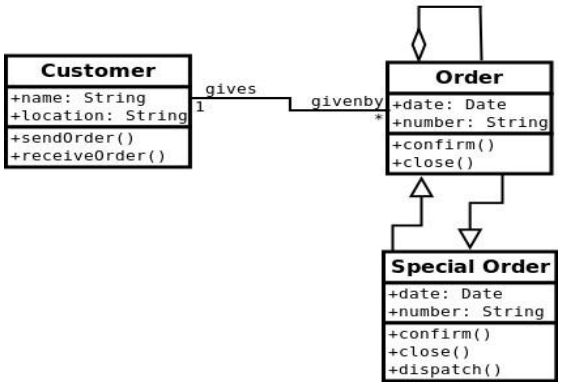


Fig.5. Class diagram of Customer Order Relationship

It can be observed form Fig. 3 and Fig. 4, that all the branches are closed. So, negation of conclusion is inconsistent with the premises, and therefore, the conclusion follows from the premise. Hence, the design rules are valid.

3.5 Case Study

In this subsection, a simple UML class diagram of Customer Order Relationship, presented in Fig. 5 is used to illustrate the approach. This diagram represents associations of Customer with Order, Order and Special Order. It consist of two classes (Customer and Order), one subclass (Special Order) of Order. It consist of following associations: Give (Customer, Order), Aggregates (Order, Order). One Generalization Relation: Special Order generalized to Order

Application of **Design Rule 1** and **Design Rule 2**, over the model presented in Fig. 5 with XML and XSD is illustrated below.

Another way of representing UML class diagram of Fig. 5 in text document form [5] is as below:

Text document:

Customer; 2; name; location;2;sendOrder;;;

receive Order;;;;1;gives;Order;0;0;0;
Order;2;date;number;2;confirm;;;close;;;;1;
aagregates;Order;0;0;0;1;SpecialOrder;
SpecialOrder;2;date;number;3;confirm;;;
close;;;dispatch;;;;0;0;0;1;Order;

XML: The Customer class of the Fig. 5 has two instance variable name and string. It has two methods sendOrder() and receiveOrder(). This class is associated with class Order. The class Special Order of the Fig. 5 has two instance variables date and number. It has three methods confirm(), close() and dispatch(). It is subclass of class Order. The complete XML document for the Fig. 5 with OCL Constraint 1 (Design Rule 1) and OCL Constraint 2 (Design Rule 2) is shown in Appendix A.

XSD: The complete XSD document for the Fig. 5 with association, generalization relationship and OCL Constraints (Design Rules) is shown in Appendix B. After validation of XML document as given in Appendix A against XSD document of Appendix B, we find that class Customer described above does not violate any design rule. While class Special Order violate design rule 2. The Appendix can be found at "http://www.mnnit.ac.in/dky" with file name "Validating Design Model.pdf".

4 Related Work

In this section the previous approaches have been discussed. These approaches can be divided into two categories, UML approaches and nonUML approaches. A short explanation of both is given starting from nonUML approaches.

4.1 NonUML Approaches

Some of the previous validation works have been done in ER conceptual schemas, mostly regarding strong satisfiability of cardinality constraints. This notion has been introduced in [6], where problem was reduced to solve a system of linear inequalities defined from the cardinality constraints. [7] used a graph theoretic approach to solve the problem, that was extended in [8] to deal the generalized concepts of cardinality.

On the other hand, in [8] graph theoretic approach is used to determine the strong satisfiability of a schema. This time the kind of constraints considered are integer cardinality constraints, which are more general in comparison to traditional one because they allow blanks in the sets of cardinalities. [9] checks satisfiability by building a minimal sample database that satisfy global cardinality constraints.

Problem of the approach provided in [10] is the identification of potentially redundant associations in an ER schema. This approach is based on adjacency matrices, but it is not complete due to existence of redundancy which involve more than one relationship between entities, that cannot be detected.

In summary, it can be said that many methods have been proposed for validation on ER schemas, mainly with cardinality constraints. None of them has dealt with design rules to validate UML model. The approach of the paper mainly focuses on design rules to validate design models which deal with some of the above problems.

4.2 UML Approaches

The satisfiability problem has also been addressed for UML schemas. UML schemas with restrictions are analyzed in [11] to find conflict related to disjointness and covering constraints in hierarchies. Another approach for reasoning on UML schema is to translate schema into description logics (DL). DL is a family of formal knowledge representation, based on first-order logic. Previously, DL has gone beyond its traditional scope in the Artificial Intelligence (AI) area to provide new options and solutions for many topics in the database and the area of conceptual modeling.

The reasoning in DL is to check the internal correctness of the UML schemas. [12] gives an approach based on this to deal with a limited set of cardinality constraints, disjointness, inclusion and exclusion dependencies.
Regarding the approaches which are not based on DLs, the problem of checking satisfiability of UML schemas has been addressed. In paper [2] an approach to verify and validate UML schemas with OCL constraints has been given. But this paper considers a specific UML schema while our focus is on analyzing design model against a set of validated design rules which are converted into predicate logic for their validation.
In [4], focus is on the finding the cause for a design model inconsistency. It is determined by validation of a design rule and design model both. This paper gives an algorithm to analyze the structure of inconsistent design rules and their behavior during validation. This paper evaluated the approach across 29 UML models against a set of twenty OCL design rules. This approach highlighted all model element properties and design rule expressions which are responsible for the generation of inconsistency. The paper uses validation tree for design rule validation. While we are trying to find inconsistencies in the design models based on the validated design rules represented through predicate logic.
Reder and Egyed in their work [13] presented an approach for generating repairs of the inconsistencies by constructing repair trees. This approach takes into account the syntactic and dynamic structure of inconsistent design rules. Paper deals with large number of repairs by focusing exactly what caused an inconsistency. Calculation of cause is done by monitoring the run time evaluation of the inconsistency to know why and where it is failed.
Article [1] validates design rules and design model based on predicate logic approach. While we have used XML approach for the validation of design model. The XML approach is easy to understand and use. One can model big system readily in XML. The same becomes difficult in logic. It is state forward to programme XML on the other hand programming logic is a cumbersome task.
The goal of [14], is to increase the performance during revalidation of design rules. It is also focused on how the memory used can be optimized for the re-validation. An approach for the incremental validation of design rules has been proposed in it. The approach is validated on 19 design rules. Our approach takes care of validation of design models as well as design rules.
An approach for reasoning on UML schema is [15], which translate a subset of UML and OCL into the Alloy logic formalization [16], with alloy analyzer. The translated subset may lack some OCL properties and association classes which are not translated. While this does not happen in our case.

5 Conclusions

A method has been presented to identify inconsistencies in the design rules as well as in design models. The focus is to validate design rules, that is achieved through an approach based on predicate logic, to find inconsistencies in the design rules. The approach consist of two steps, first is the translation of OCL constraints (Design Rules) into their normalized form (in terms of select and size OCL operations) and second step is the conversion of normalized OCL constraints into predicate logic representation. Further, we have applied tableaux method for finding inconsistencies in the design rules. Valid design rules are used for validating design models. An approach based on XML has been used for validation of a UML model.
In future, plan is to explore the work for automatic validation. Further, we will extend the work to include more OCL design rules. The effort will also be in finding the dependencies of one design rules over other rules. These dependencies will be represented by dependency graph. The dependency graph may be used for finding cyclic dependencies among design rules. To avoid infinite sequence of repairs, the elimination of cyclic dependency is required.

References

[1] Ashish Kumar Mishra and Dharmendra K. Yadav. Validation of UML design model. JSW, 10(12):1359–1366, 2015.

[2] Anna Queralt and Ernest Teniente. Verification and validation of uml conceptual schemas with ocl constraints. ACM Transactions on Software Engineering and Methodology (TOSEM), 21(2):13, 2012.

[3] J. Kelly. The Essence of logic. Pearson Education India.

[4] Alexander Reder and Alexander Egyed. Determining the cause of a design model inconsistency. IEEE Transactions on Software Engineering, 39(11):1531–1548, 2013.

[5] MAPPING UML DIAGRAMS TO XML. http://uml2xml.tripod.com.

[6] Maurizio Lenzerini and Paolo Nobili. On the satisfiability of dependency constraints in entity-relationship schemata. Inf. Syst., 15(4):453–461, 1990.

[7] Bernhard Thalheim. Entity-relationship modeling - foundations of database technology. Springer, 2000.

[8] Sven Hartmann. On the consistency of int-cardinality constraints. In Conceptual Modeling ER '98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings, pages 150–163, 1998.

[9] Konrad Engel and Sven Hartmann. Minimal sample databases for global cardinality constraints. In Foundations of Information and Knowledge Systems, Second International Symposium, FoIKS 2002 Salzau Castle, Germany, February 20-23, 2002, Proceedings, pages 268–288, 2002.

[10] David S. Bowers. Detection of redundant arcs in entity relationship conceptual models. In Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Tampere, Finland, October 7-11, 2002, Proceedings, pages 275–287, 2002.

[11] Ken Kaneiwa and Ken Satoh. Consistency checking algorithms for restricted UML class diagrams. In Foundations of Information and Knowledge Systems, 4th International Symposium, FoIKS 2006, Budapest, Hungary, February 14-17, 2006, Proceedings, pages 219–239, 2006.

[12] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. Artif. Intell., 168(1-2):70–118, 2005.

[13] Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, pages 220– 229, 2012.

[14] Alexander Reder and Alexander Egyed. Incremental consistency checking for complex design rules and larger model changes. In Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings, pages 202–218, 2012.

[15] Kyriakos Anastasakis, BehzadBordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to alloy. Software and System Modeling, 9(1):69–86, 2010.

[16] Daniel Jackson. Alloy: a lightweight object modelling notation. ACM Trans. Softw.Eng. Methodol., 11(2):256–290, 2002.