Validation of UML Design Model

Ashish Kumar Mishra, Dharmendra K. Yadav*
The Department of Computer Science and Engineering, MNNIT Allahabad, Allahabad.

* Corresponding author. Tel.: 08004486303; email: dky@mnnit.ac.in
Manuscript submitted June 5, 2015; accepted August 30, 2015.
doi: 10.17706/jsw.10.12.1359-1366

Abstract: Ensuring absence of design model inconsistencies that may lead to errors creeping in software, is a major challenge of software development. Further, these errors may propagate to the different phases of software development life cycle models. Correct design rules help in developing consistent design models of software. A way to ensure the correctness of design models is through its validation. Validation of design models is based on validated design rules. This work proposes a method based on predicate logic for validation of design models. The method has been illustrated with the help of a verified case.

Key words: Consistency, constraints, design models, design rules, OCL, predicate logic, UML.

1. Introduction

In this paper, a method based on predicate logic is presented to determine the inconsistency in the UML design models. Modeling languages for design such as UML, follows various constraints. Constraints have been used for the various purposes such as representing modeling philosophies, reflecting restrictions of domain and application needs as being part of UML. In this paper, we use the term design rule to denote such constraints. A design rule consists of constraints that validate to either true (consistent) or false (inconsistent) to reflect whether the model satisfies given rules or not.

An inconsistency is a sign of problem and the element that causes the inconsistency needs to be resolved. So, all model element's properties that causes inconsistency must be identified to address them. An accurate visualization of the cause can contribute to a better design process, which avoid further errors. The proper understanding of the cause may help to focus on the repair of those parts of the model which contributed to the inconsistency.

This paper focuses on the problem of validating design rules for UML design models. Design model's properties may be specified with design rules. These design rules need to be formalized, if they are required to be validated. Design rules can be expressed through OCL (Object Constraints language). OCL constraints are converted into predicate logic for the validation of design rules. Predicate logic expressions of design rules are validated through tableaux method. Predicate logic has been used for validation of model as it is rigorous language and captures models precisely. Validation of UML design models using design rules concerning re-usability is the main goal of the work.

2. Basic Concepts

In this section the definitions of the concepts used in the paper are provided. In section 2.1 the concepts regarding UML class diagrams and OCL constraints are defined. Section 2.2 introduces the tableau method.

2.1. Basic Concepts on UML and OCL

The UML has become a de facto standard for the conceptual modeling of information systems. In UML, a conceptual schema (CS) is represented by a class diagram. A class diagram consists of set of classes, associations, and integrity constraints that can be expressed graphically. Additionally, the conceptual schema must include all relevant integrity constraints definitions. The constraints that cannot be expressed graphically must be expressed by languages such as OCL (Object Constraint Language).

OCL is the expression language for the UML. It has the features of an expression language, a modeling language and a formal language. The OCL is a part of the UML specification. In object oriented modeling, a graphical model, such as a class model, is not sufficient for an accurate and unambiguous specification. There is a need of additional constraints for the objects in the model. Such constraints are often described in natural language that may result in ambiguities. Hence, formal languages have been developed to write unambiguous constraints. We have selected OCL over other formal languages because it is very easy to use and capture design rules lucidly.

2.2. Description of Tableaux Method

As validation of arguments is done through logic. A method for showing validity of arguments is by truth table. The issue with validation through truth tables is that they get very large rapidly. In general, a truth table with n variables has 2n rows causing exponential explosion.

The use of tableaux method is based on the strategy to negate the conclusion of an argument for checking its consistency with the premises. The idea is to check whether it is possible for a false conclusion to be consistent with true premises. If it is not possible i.e., the conclusion must be true when the premises are true, it is called that conclusion is semantically entailed by the premises.

A semantic tableau is a sequence of formulae constructed according to certain rules, and usually organized in the form of a tree. The rules are shown in Fig. 1. Where t is a term in Rule10 which is known as universal instantiation (UI). And in Rule 11, t1 is a term which has not been used in the derivation so far.

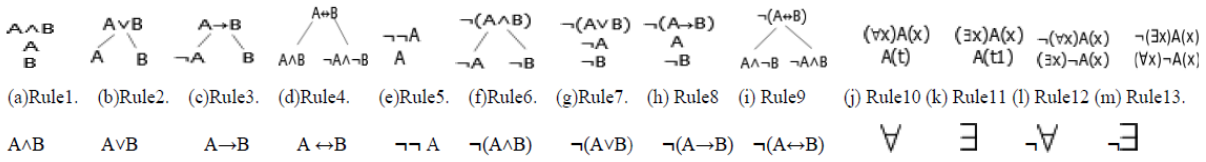

Fig. 1. Rules for semantic tableau.

Rule (14). When A and ¬A appear in a branch of tableau, inconsistency is indicated and it is not further extended i.e. it is closed. If the tableau is closed, statements in the original arguments are consistent.

3. Validation of Design Rules

If three OCL design rules of [1] have been translated into predicate logic formulae using the approach provided in [2]. Summary of the approach given in [2] is also presented in section 3.1. These design rules are basically concerned with re-usability of software. Further, these design rules expressed in predicate logic have been validated by semantic tableaux method of the logic. Section 3.2 provides the list of three OCL design rules. Section 3.3 presents the detail steps for conversion of these design rules into predicate logic. In section 3.4 the validity of the design rules in predicate logic by semantic tableaux method is presented. A case study is shown in Section 3.5.

3.1. Approach for Logic Formalization

To validate UML design rules in OCL forms, we translate them in to predicate logic representation. Then the problem is reduced to checking the consistency of these predicates. The translation of OCL constraints into predicate logic involves two steps. First step is the transformation of each OCL expression into equivalent expression in terms of select and size. Select and size OCL operations are applied to collections of elements, size returns the number of elements in the collections and select returns the subset of the collection which satisfies the condition. Table 1 shows the OCL operations and gives their equivalent simplified (normalized) expressions.

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→exists(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<>e2 and e2.body=e.body))→size()=0
path→one(e|body)	path→select(e|body)→size()=1
path→reject(e|body)	path→select(e|not body)

Once simplified (if needed) the original invariant, which we have now have a limited set of patterns. In the following we denote the original expression by expr and its translation into logic by *Tr*(expr). We assume path=obj0.*r*1.. . . *rn* is a path starting from obj of a class *C*, navigating through roles *r*1 to *rn*, where *rn* is a role or an attribute, resulting in a set of objects or values.

a) expr = path

Tr(expr) = *c*(Obj0)∧ assoc1(. . . , Obj0 , . . . , Obj1 , . . . , Objk.)∧ . . .∧assocn(. . . , Objn−1 , . . . , Objn, . . . , Objl) where assocj(. . . ,Objj−1 ,. . . ,Objj,. . . ,Objk) is the logic representation of the association of arity *k*, with or without an association class, between roles *rj*−1 and *rj*. The predicate assocn is binary if it represents an attribute.

b) xpr = 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) [∧*Tr* (value)]∧ Obj1 opComp Obj2

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* (obj0 .*r*1 . . . *rn*[→select(expr)]→size()<*k*)= *Tr* (obj0)∧¬cond(Obj0, *X*, . . . , *Xm*)
cond(Obj0, *X*, . . . , *Xm*)←*Tr*1(path)[∧ *Tr*1(expr)]∧. . .∧*Trk*(path) [∧ *Tr k*(expr)]∧ Obji <> Objj

(c.2)*Tr*(obj0.*r*1 . . . *rn*[→select(expr)]→size()>*k*)= *Tr*1(path)
[∧*Tr*1(expr)]∧.. .∧*Trk*+1(path) [∧*Trk*+1(expr)] ∧Obji <> Objj

(c.3)*Tr*(obj0.*r*1. . . *rn*[→select(expr)]→size()=*k*)= *Tr*1(path)
[∧ *Tr*1(expr)]∧. . .∧*Trk*(path)[∧*Trk*(expr)]∧ ¬cond(Obj0, *X*,. . . ,*Xm*)
cond(Obj0, *X*,. . . , *Xm*)←*Tr*1(path)[∧*Tr*1(expr)]∧. . .∧*Trk*+1(path)[∧*Trk*+1(expr)]∧Obji <> Objj

(c.4)*Tr*(obj0.*r*1.. . *rn*1.*rn*[→select(expr)]→size()<>*k*)= *Tr*(obj0)∧¬cond(Obj0, *X*,. . . , *Xm*) cond
(Obj0,*X*, ..., *Xm*)←*Tr*1(path)[∧*Tr*1(expr)]∧ . . . ∧*Trk*(path)[∧*Trk*(expr)]
∧Obji<>Objj∧¬oneMore(Obj0, *X*, . . . ,*Xm*) one More(Obj0, *X*,. . . ,*Xm*)←*Tr*1(path)[∧*Tr*1(expr)]

∧.. .∧*Trk*+1(path)∧*Trk*+1(expr)]∧ Obji <> Objj

where Obj0 represents the object at the starting of the path, and Obji represents an object obtained as a result of the navigation path according to Tri(path). Each expression Tri refers to the same translation but using different variables, for all terms different from the initial object Obj0.

d) expr = expr1 and expr2 Then, *Tr*(expr) = *Tr*(expr1)∧ *Tr*(expr2)

e) expr = expr1 or expr2 Then, *Tr*(expr)=disjunction(Obj0, *X*, . . . , *Xm*)

where disjunction is a derived predicate defined by the rules:

disjunction(Obj0, *X*, . . . , *Xm*)←*Tr*(expr1)

disjunction(Obj0, *X*, . . . , *Xm*)←*Tr*(expr2)

f) expr = not expr, In this case, Tr(expr) depends on the kind of expression:

(f.1) *Tr*(not path[[→select(expr)]→size()] opComp value)=*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.

3.2. Design Rules

The three UML design rules with their OCL form taken from [1] are given 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 : A Classifier may not declare an Attribute that has been declared in Parent Classifiers.

OCL Form : context Class inv : self.allParents()→forAll(classifier|c.oclAsType(Class). ownedAttribute→forAll(p:Property|p.class.ownedAttribute→collect(name)→excludes(p.name)))

Design Rule 3 : No circular Inheritance allowed.

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

3.3. Logic Conversion of Design Rules

The detail steps for conversion of the design rules into predicate logic is presented below:

Design Rule 3 : Logic Conversion:

let expr=not self.allParents()→includes(self) **(from Table I)**

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

Validity of the design rules by Tableaux method is illustrated below. These rules can be validated by negating the conclusion and taking logical AND of it with the premises. If all the branches of the tableaux are closed, we can infer that negation of conclusion is inconsistent with the rule. Hence, the conclusion is con- sistent with rule.

Design Rule 3 in logic form after replacing values of derived predicate:

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

Invariant of design rule 3 is 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 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. 2.

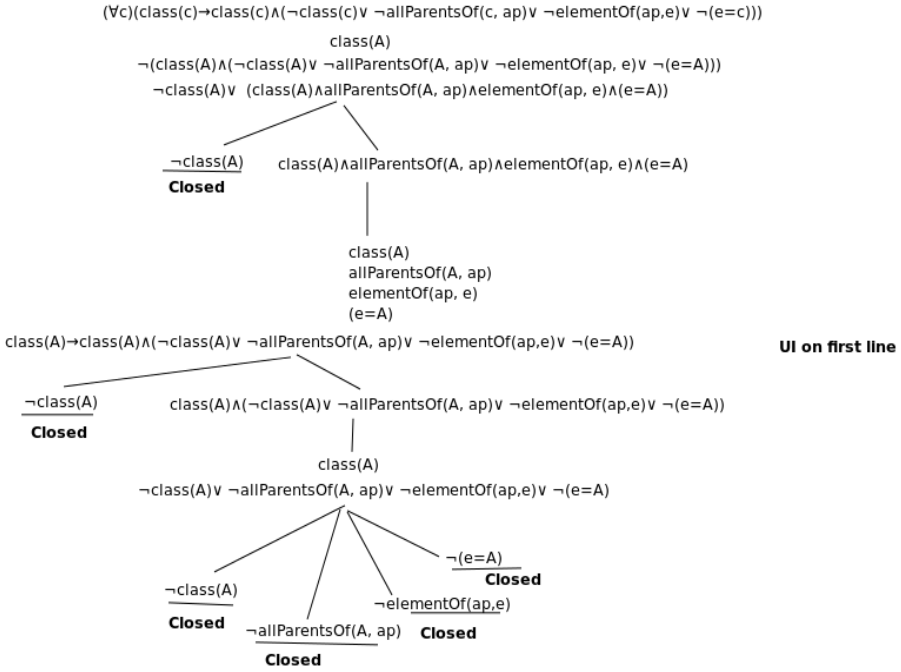

Fig. 2. Closed tableau of design rule 3 with negation of conclusion

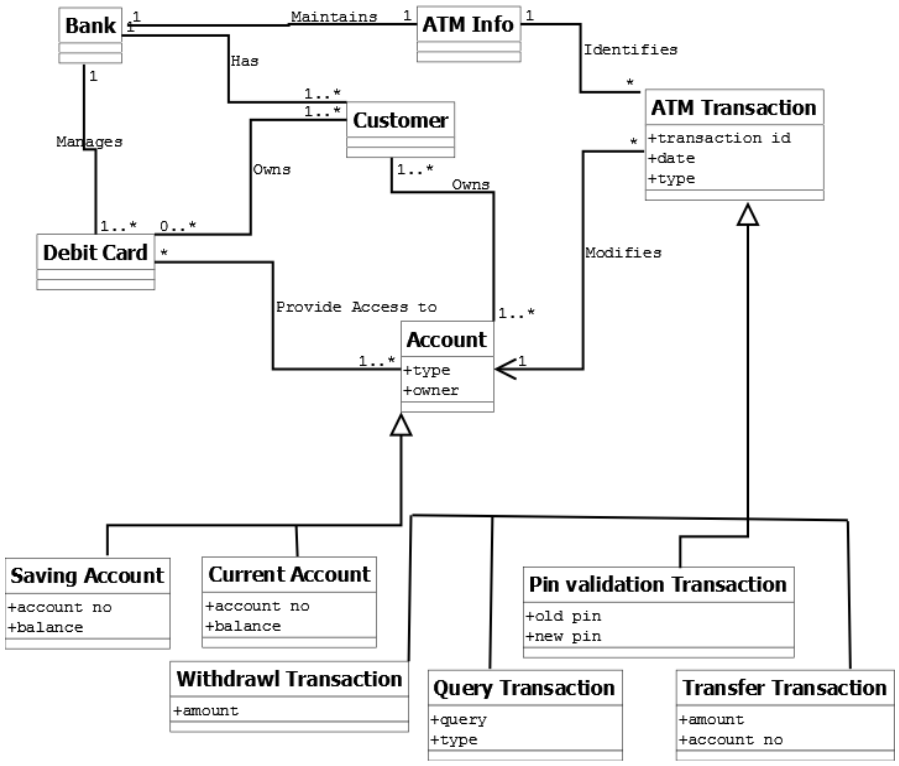

Fig. 3. Class diagram of ATM.

We can observe in Fig. 2, that all the branches are closed so negation of conclusion is inconsistent with the premises, and therefore, the conclusion follow from the premises. So the design rules is valid. Similarly we can do for other two rules also.

3.5. Case Study

In this section, a simple UML class diagram of ATM, presented in Fig. 3 is used to illustrate the approach. This diagram represents associations of ATM with customer, Bank and Account. It consist of six classes (Bank, ATM Info, Debit Card, Customer, Account and ATM Transaction), two subclasses (Current Account, Saving Account) of Account and four subclasses (Withdrawal Transaction, Query Transaction, Transfer Transaction, and Pin validation Transaction) of ATM Transaction. It consist of following associations: Maintains(Bank, ATM Info), Has(Bank, Customer), Manages(Bank, Debit Card), Owns(Debit Card, Customer), Owns(Customer, Account), Modifies(ATM Transaction, Account), Provide Access to(Debit Card, Account), Identifies(ATM Info, ATM Transaction). Two Generalization Relation: Current Account and Saving Account generalized to Account, Withdrawal Transaction, Query Transaction, Transfer Transaction and Pin validation Transaction generalized to ATM Transaction.

Application of Design Rule 3 over the model is presented in Fig. 3. Design Rule 3:

Tr(expr)=class(c) /{¬class(c) \/¬allParentsOf(c,ap) \/¬elementOf(ap,e) \/¬(e=c))

Invariant in Design Rule 3 is for class, so we have to check this rule for all classes (12 here).

Check for class Saving Account

Saving Account class has parent class Account which is different from context class

(Saving Account), so *Tr*(expr)=class(Saving Account) /{¬class(Saving Account) \/

¬allParentsOf(SavingAccount,Account) \/¬elementOf(Account, Account) \/¬(Account=Saving Account))

Since Account and Saving Account are two different class, so

Tr(expr)=T /{¬T \/¬T \/¬T \/¬F) Since ¬T=F, hence, *Tr*(expr)=T /{F \/F \/F \/T)

Since T \/F=T, hence, Tr(expr)=T /{T) Since T /T=T, hence, *Tr*(expr)=T

So Design Rule 3 is satisfied for class Saving Account. Similarly we can show that Design Rule 3 is satisfied for all remaining classes. So it can be concluded for the model presented in Fig. 3, that all associations satisfy Design Rule 1, all classes except Query Transaction satisfy Design Rule 2, and all classes satisfy Design Rule 3.

4. Related Work

The work is mainly related with UML schemas, that are based on description logics (DL). DL is a family of formalisms for knowledge representation, based on first-order logic. In the past, DL has gone beyond its traditional scope in the Artificial Intelligence (AI) area to provide new options and solutions to many topics in the database and conceptual modeling areas.

Regarding those approaches that are not based on DLs, the problem to check satisfiability of UML schemas has been addressed. Restricted UML schemas are analyzed. in [3], for consistency checking. The approach to validate an UML schema is the translation of UML schema into DL to perform several validation task using a DL-based system. In paper [2] an approach to verify and validate UML conceptual schema with OCL constraints has been provided. But this paper considers a specific UML schema while our focus is on analyzing design model built through validated design rules which are converted into predicate logic.

In [1], the cause of a design model inconsistency is determined by validation of a design rule. This paper analyzes the structure of inconsistent design rules and their behavior during validation and also presents an algorithm. The approach was evaluated across 29 UML models against a set of 20 OCL design rules. This approach lists all model element properties and design rule expressions that are responsible for the inconsistency. This paper uses validation tree for validation of design rules. While in our approach we are

trying to find inconsistencies in the design models based on the validated design rule represented through first order predicate logic.

Reder and Egyed in their work [4] discuss the approach for repairing the inconsistencies by constructing repair trees. This paper deals with large number of repairs by pin-pointing on what caused an inconsistency. It presents repairs as a linearly growing repair tree. The cause is calculated by testing the run time evaluation of the inconsistency to know where and why it is failed. The repair tree modeled individual changes that make repairs as alternatives and sequences representing the syntactic structure of the inconsistent design rule. Our approach deals with finding inconsistencies in the design models.

In [5], it is shown how the performance and the memory used for the re-validation of design rules can be optimized. This paper proposes an approach for the incremental validation of design rules. This approach is applied on 19 design rules. This work improves the way for processing a larger number of changes in model and/or more complex model changes with instant response times. Our approach takes care of validation of design models.

5. Conclusions

An approach has been presented for identifying inconsistencies in the design rules and design models. We have validated design rules using a method based on predicate logic to find inconsistencies in the design rules. In this approach we have first translated OCL constraints into their normalized form further converted them into predicate logic representation. We applied tableaux method to find inconsistent design rules. Further we have used the validated design rules for ensuring the consistency of a UML design model.

In future automatic validation of the approach may be explored. Further, the work may be extended to include more design rules or OCL expressions. The effort may be towards finding the dependency of design rules over one another. Dependency of one rule on the other will be represented by dependency graph. Use of dependency graph to find cyclic dependency among design rules may be explored. This cyclic dependency may be eliminated to avoid infinite sequence of repairs.

References

[1] Alexander, R., & Alexander, E. (2013). Determining the cause of a design model inconsistency. *IEEE Transactions on Software Engineering, 39(11),* 1531–1548.

[2] Anna, Q., & Ernest, T. (2012). Verification and validation of uml conceptual schemas with OCL constraints. *ACM Transactions on Software Engineering and Methodology (TOSEM), 21(2),* 13.

[3] Ken, K., & Ken, S. (2006). Consistency checking algorithms for restricted UML class diagrams. *Proceedings of the 4th International Symposium, FoIKS, Budapest, Hungary, Foundations of Information and Knowledge Systems* (pp. 219–239).

[4] Alexander, R., & Alexander, E. (2012). Computing repair trees for resolving inconsistencies in design models. *Proceedings of the International Conference on Automated Software Engineering* (pp. 220–229).

[5] Alexander, R., & Alexander, E. (2012). Incremental consistency checking for complex design rules and larger model changes. *Proceedings of the 15th International Conference Model Driven Engineering Languages and Systems* (pp. 202–218).

Ashish Kr. Mishra received his M.Tech. degree in computer science and engineering from Motilal Nehru National Institute of Technology, Allahabad, India, in 2015. He is working as an assistant professor in the Department of Computer Science and Engineering, LDC Institute of Technical Studies, Allahabad, India. His research interests include areas in object

oriented modeling, software engineering, and formal methods. He has 2 International publications in various Conferences and Journals.

D. K. Yadav received his PhD from IIT Bombay, Mumbai, India and his M. Tech. degree in computer science and engineering from Motilal Nehru National Institute of Technology, Allahabad, India. He completed his M. Tech thesis from BEA Systems Sanfrancisco, CA, USA. He is working as an associate professor in the Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology, Allahabad, India. His research interests include areas in software architecture, service oriented software engineering, web services and formal methods. He has 10 International publications in various conferences and journals.