Verifying Static Aspects of UML models using Prolog

Feng Sheng Huibiao Zhu* Zongyuan Yang Jiaqi Yin Gang Lu*
School of Computer Science and Software Engineering
East China Normal University, Shanghai, China

***Abstract*—The Unified Modeling Language (UML) provides a number of diagrams to describe the modeling system from different perspectives, which contain overlapping information about the systems. However, it does not provide any means of meticulously checking consistencies among the overlapping elements. In this study, we propose an approach for consistency checking of UML class diagrams and object diagrams using Prolog. First we formalize the model elements based on metamodel and convert the models into Prolog facts. Then we define some consistency rules that are encoded into Prolog. The Prolog's reasoning engine automatically checks the consistencies of models. In addition, we provide interfaces to query models for properties, elements and submodels. The design errors can be effectively avoided and the correctness of code-generalization can be guaranteed according to our approach.**

***Index Terms*—Class Diagram, Object Diagram, Consistency Checking, Prolog**

I. Introduction

The Unified Modeling Language [1] has been developed as a standard object-oriented modeling notation in Model Driven Engineering (MDE) and is widely used in industry. It provides numbers of diagrams to model different aspects of the systems, such as the static views (class diagrams, object diagrams) and dynamic views (sequence diagrams, statecharts). In addition, it offers a variety of tools that cover all the features of the system modeling for a more complete description of the models. However, the syntax and semantics of the UML are semi-formally defined in terms of a metamodel combing natural language descriptions, UML notations and Object Constraint Language (OCL), which is not sufficient to express the semantics of the UML models precisely. Moreover, UML uses the different models to characterize the same system from different perspectives. Change in one diagram may ultimately affect the other diagrams, and result the inconsistencies between different diagrams.

UML models can be represented in the form of a theory in mathematical logic, such as the description logic [2], data refinement [3] and category theory [4]. By transforming UML models into mathematical logic, the problem of inconsistencies can be regarded as the problem of contradictions in the logic theory. A system is consistent if it does not contain any contradictions. Inconsistencies in UML models reveals design errors in software development. Moreover, the inconsistencies will not be propagated to the codes if we perform the consistency checking at design phase.

Various approaches [6] [13] [15] [16] have been proposed to check the consistency in UML. Typically, approaches devoted to the verification of UML models convert the models into formal semantic domains. Besides, different types of consistency rules are defined on the basis of these conversions. The models' consistencies are verified according to the possibility that the models satisfy these consistency rules. Straeten et al. [2] developed an extension of UML metamodel and presented a classification of inconsistency problems using description logic. They expressed the detection and resolution of consistency conflicts by means of rules. Egea and Rusu [11] demonstrated the structural and semantic conformance between models and metamodels by transforming the models into Maude. Besides, the satisfiability modulo theories (SMT) is often used to support the consistency verification of UML models. Soeken et al. [8] presented an automatic approach that checks the verification for the UML dynamic models. The underlying verification problem is encoded as an instance of the satisfiability problem and subsequently solved using a SAT Modulo Theory solver. However, the SMT solvers typically only support decidable theories and are not sufficient for consistency checking. Storrle [12] proposed a representation of models based on Prolog. He provided query interfaces to identify elements, properties and submodels. Khai et al. [10] proposed an approach for consistency checking of class and sequence diagrams based on Prolog. Cabot et al. [21] presented an automatic method for the verification of UML class diagrams extended with OCL constraints. They transformed the UML/OCL model into a Constraint Satisfaction Problem. The correctness properties such as weak and strong satisfiability or absence of constraint redundancies are checked. Khan and Porres [7] proposed an approach to automatically validate the consistency of UML models using logic reasoners for the Web Ontology Language OWL 2. They translated the models into OWL 2 and presented a tool supporting UML modeling tools to perform the translation from the models into the OWL 2.

In this paper, we propose an approach for the fully automatic, expressive verification of UML class diagrams and object diagrams using Prolog. First we formalize the model elements based on metamodel and convert the models into Prolog facts. Then we summarize several different types of consistency problems and encode into Prolog rules. Finally the

*Corresponding authors: hbzhu@sei.ecnu.edu.cn (H. Zhu).
glu@cs.ecnu.edu.cn (G. Lu).

DOI reference number: 10.18293/SEKE2019-175

reasoning engines such as SWI-Prolog, are used to check the inconsistencies through analyzing the models and feedback the error information if any inconsistency error occurs. According to our approach, the design errors can be effectively avoided in design phase and the correctness of code-generalization can be guaranteed.

This paper is structured as follows. In Section II, we introduce the background about model consistency and our approach. Section III presents the formalization and conversion from UML models to Prolog. Section IV describes the different types of consistency. We have implemented a prototype tool which automatically translates the UML models into Prolog codes in Section V. Section VI shows some experiments to indicate the performance of our approach. Section VII concludes the paper and discusses the further work.

II. Background

A. Consistency checking of UML models

In the past few years, the consistency problems in UML have become a hot issue. The term *model consistency* [18] is defined as "the overlapping elements in different models of the same system satisfy certain properties". There are many studies that classify the consistency of models. One of the most widely accepted classifications is the Engels' classification [6], which classifies the model consistency into four categories:
(1) *Vertical consistency* occurs when an abstract model is refined into a more concrete model. It is desirable that the concrete model should be consistent with the abstract one. The refinements of models cause the vertical consistency problems.
(2) *Horizontal consistency* arises in case where different models describe the same system from different aspects containing overlapping elements. The overlapping elements should satisfy some elementary properties to ensure the consistency between the different models.
(3) *Syntactical consistency* ensures that a model conforms to the abstract syntax. The abstract syntax of models is usually defined by the metamodel. In other word, we consider that the models are syntactically consistent if the models are the instances of classes and associated by the instances of associations in the metamodel.
(4) *Semantic consistency* occurs when the developers expect to get more accurate models through additional constraints on models. The semantics of UML is usually specified in natural languages and OCL. It is hard to check the semantic consistency in UML especially for static diagrams since the OCL is not precisely defined in UML.

In this paper, we mainly consider three kinds of consistency issues: horizontal, syntactical and semantic consistency. The vertical consistency involving the refinements of the models is out of the scope of this paper.

B. Overview of the approach

We propose a general framework to check the consistency for a subset of UML static diagrams including class diagrams and object diagrams. The basic route of our approach is shown as Fig. 1. First the designer provides a target model, created by UML CASE tools, and transforms the models into XMI files. Then the concepts of the models are automatically converted to the facts in Prolog database. Next the consistency rules with the facts are imported into the SWI-Prolog. The SWI-Prolog analyzes and queries the elements to verify the consistency of the target model and feedback the error information if any inconsistency occurs.

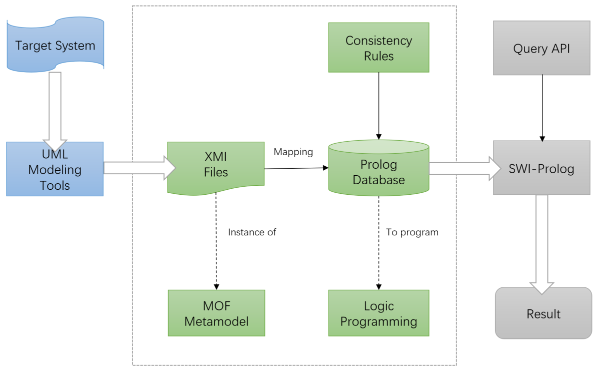


Fig. 1. An Outline of Our Approach

III. From UML Models to Prolog

A. From Class Diagrams to Prolog

The metamodel is a language that contains certain metadata describing the concepts and relations for providing a modeling language. The *Meta Object Facility* (MOF) standard defines the Essential MOF (EMOF), a subset of MOF that is used to define the metamodels. In this study, we formally define the syntax of the models according to the EMOF metamodel.

Definition 1: The syntax of the class diagrams is a structure

$$\mathcal{M} = \{class, attribute, operation, association, \\ rolename, multiplicity, \prec\}.$$

where

- *class* is a set of classes.
- *attribute* is a set of signatures for functions mapping a class c to an associated attribute value.
- *operation* is a set of signatures for user-defined operations of a class c.
- *association* is a set of associations.
- *rolename* is a set of roles.
- *multiplicity* is a set of multiplicities of associations.
- $\prec$ is a partial order on classes reflecting the generalization hierarchy of classes.

The class diagrams are denoted as a series of Prolog facts. Each model element in $\mathcal{M}$ is described as a set of facts with the same clause and different parameters. Every element of class diagrams is assigned an identifier that identifies the actual objects that are to be instances of the various classes in the metamodel. In the following, each model elements is considered in details.

The most important part of the class diagrams is the classes. A class is a collection of objects that have the same attributes, operations, relationships, and semantics. The `class/3` clause in Prolog is used to denote the class, including an identifier, a class name and a boolean type indicating whether it is abstract.

$$class(classid, classname, isAbstract).$$

Note that the abstractions of the objects are classes, and the instantiations of the class are objects. The classes can be considered as the types of objects according to the Model Driven Architecture (MDA).

The classes define a group of objects' states and behaviors. More specifically, the attributes and associations of classes define the objects' states and relationships respectively, and the operations describe the behaviors of objects. The attributes are the values describing the object's properties, including a name and a type that specifies the domain of values. The `attribute/4` clause describes the attributes of class diagrams in Prolog.

$$attribute(attrid, attrname, attrtype, classid).$$

where `attrid` is an identifier of attribute, `attrname` is an attribute name, `attrtype` is the type of attribute, and `classid` is an identifier of class to which this attribute belongs.

The operations are parts of a class declaration in models. They describe the behavioral properties of classes, represented by the `operation/4` clause in Prolog.

$$operation(opid, opname, [parameters], classid).$$

where `opid` is an identifier of the operation, `opname` is a name of the operation, `[parameters]` is a list of parameters' id, and `classid` is an identifier of class to which this operation belongs.

The `parameter/3` clause in Prolog denotes the parameters in operations.

$$parameter(pid, pname, ptype).$$

where the `pid` and `pname` denote the identifier and name of the parameter respectively, and `ptype` denotes the type of the parameter, including the primitive types and class types.

The associations describe the structural relationships between the classes. In general, a class can have more than one associations, and an association can connect two or more classes. In this study, only binary associations are considered, the *n-ary* associations can be obtained by extending parameters in the `association/4` clause, where the `assoctype` can be directional, nondirectional, aggregate or compositive.

$$association(associd, classAid, classBid, assoctype).$$

In an association, the class can appear more than once playing different roles. The role names are usually useful in the navigations of models. The `rolename/4` clause assigns a unique role name to each class that participates in the binary association. The order of names in role names should coincide with the order of classes in the corresponding associations. If the role names are omitted in a class diagram, we define the role names by changing the first letter of the name of target class to lower case.

$$rolename(roleid, nameA, nameB, associd).$$

Associations may also have multiplicities which specify the possible numbers of links for associated classes. The multiplicity describes the number of allowable objects of a range class to link with the objects of a domain class. The `multiplicity/5` clause has a minimum and maximum number of instances of the target classes, defined by the *lowval* and *upval* attributes. Unbounded ranges can be modelled using the value n for the upper attribute in Prolog.

$$multiplicity(multid, classid, lowval, upval, associd).$$

A generalization indicates that one of the two related class (subclass) is considered to be a specialized form of the other (superclass). The superclass is a generalization of the subclass. Generalization relationships form a hierarchy over the set of classes. A generalization hierarchy $\prec$ is a partial order on the set of classes, shown as the `generalization/3` clause.

$$generalization(genid, subid, superid).$$

We define a recursive function *parents* to get all superclasses of a given class as follows.

$$parents : \begin{cases} class \to \mathcal{P}(class) \\ c \mapsto \{c' \mid c' \in class \wedge c \prec c'\} \end{cases} \tag{1}$$

The `parents/2` clause indicates the parent classes can be directly or indirectly derived through the generalization relationships. We can query the parent classes using `all_parents/2` where `findall` get all parent classes from the `parents` and put them into the variable `IDS`.

```
parents(Superid, Subid) :-
  generalization(_, Subid, Superid).
parents(Superid, Subid) :-
  generalization(_, Subid, X),
  parents(Superid, X).
all_parents(Subid, IDS) :-
  findall(Y, parents(Y, Subid), IDS).
```

B. *From Object Diagrams to Prolog*

The object diagrams show a complete or partial view of the structure of a modelled system at a specific time. The object diagrams is composed of the snapshots of running systems. An instance of a class is called an *object*, whereas an instance of an association is called a *link* that is a connection between two or more objects of classes at corresponding positions in the association. The object diagram focuses on the set of objects and attribute values, and the links between these objects at a particular time.

Definition 2: An object diagram for a model $\mathcal{M}$ is a structure

$$\sigma(\mathcal{M}) = \{object, link, attrval\}.$$

where

- *object* is a set of objects.
- *link* is a set of links connecting objects.
- *attrval* is a set of functions assigning attribute values to each object.

The representation of the object diagrams in Prolog is similar to the representation of class diagrams.

$$object(objid, objname, classid).$$
$$link(linkid, objAid, objBid, associd).$$
$$attrval(attrvalid, attrid, value, objid).$$

The finite sets of `object/3` clauses contain all objects and the finite sets of `link/4` clauses contain links connecting objects. The `attrval/4` clauses assign attribute values to each object. The `classid` in `object/3` should be related to the `classid` in `class/3`, the same as `associd` and `attrid`.

Definition 3: The domain of a class is defined as a set of object identifiers that are the instances of the class.

$$domain(c) = \bigcup\{objects(c') \mid c' \in class \wedge c' \prec c\}. \quad (2)$$

where the function *objects* gets all the objects of a given class. The domain of a class is defined using recursive predicate in Prolog. The `all_objects_ids/2` returns the list of objects' identifiers for a given class identifier.

```
objects_ids(Classid, Objid) :-
  object(Objid, _, Classid).
objects_ids(Classid, Objid) :-
  generalization(_, Z, Classid),
  objects_ids(Z, Objid).
all_objects_ids(Classid, IDS) :-
  findall(Y, objects_ids(Classid, Y), IDS).
```

IV. Consistency Checking Rules

A. Syntactical Consistency

A metamodel defines the abstract syntax of the models. The models are syntactically consistent if they conform to the metamodel. More specifically, the elements of the model should be the instances of the classes in the associated metamodel, and the links of two elements are the instances of associations related by the associated classes in the metamodel. The MDA is described in four-layer architecture, each layer model can be regarded as instances of the upper layer model. The representation of UML concepts in this study is based on the EMOF metamodel. The models defined in our approach satisfy the syntactical consistency since they are the instances of the metamodel.

B. Semantic Consistency

A metamodel may also define a set of validity constraints on the metamodel using OCL, called semantic consistency [19]. For instance, a class should not define two attributes having same names. These OCL constraints are defined between the metamodels and models in order to describe more detailed models. In a sense, any OCL expressions can be converted to the Prolog rules having the same semantics. The models are semantically consistent if the models conform to the rules. First the OCL representation is presented, and then we give the Prolog code for these constraints. Only several common constraints are presented because of the limited space.

Name Unique. The names of classes and associations should be unique and the names of attributes are unique in one class.

```
context Class inv:
  self.attribute -> forall(a1, a2 : attribute|
  a1 <> a2 implies a1.name <> a2.name).
```

Acyclic Generalization. There are no direct or indirect cycles in the generalization relationship.

```
context Class inv :
  self.allParents -> excludes(self).
```

The `list_reps/2` shows the repetitions in the list. The list of names is the first parameter and the result of repetition elements is the second parameter. The circular generalization error occurs if any class is in the list of its parents classes. The `inheritSelf` returns true if the parameter class `X` has cycles in the generalization relationship.

```
list_reps([],[]).
list_reps([X|Xs],Ds1) :-
   x_reps_others_fromlist(X,Ds,Os,Xs),
   list_reps(Os,Ds0),
   append(Ds,Ds0,Ds1).
x_reps_others_fromlist(_X,[],[],[]).
x_reps_others_fromlist(X,[X|Ds],Os,[X|Ys]) :-
   x_reps_others_fromlist(X,Ds,Os,Ys).
x_reps_others_fromlist(X,Ds,[Y|Os],[Y|Ys]) :-
   dif(Y,X),
   x_reps_others_fromlist(X,Ds,Os,Ys).
canGoTo(X, N, Nodes) :-
  member(X2, [X|Nodes]),
  generalization(_, X2,X1),
  \+ member(X1, Nodes),
  canGoTo(X, N, [X1|Nodes]).
canGoTo(_, N, N).
canGoTo(X, Nodes) :-
  canGoTo(X, Nodes, []).
inheritSelf(X) :-
  canGoTo(X, Nodes), member(X, Nodes), !.
```

We also offer query interfaces to query properties in SWI-Prolog. For example, the variable `List` indicates the set of the classes' name and the variable `R` represents the repetition elements if there are any repeating elements in the models.

```
?- bagof(Y,X^class(X,Y,_),List), list_reps(List,R).
```

C. Horizontal Consistency

The problems of horizontal consistency is caused by the overlapping elements of different diagrams. There are some overlapping elements between the class diagrams and object diagrams. For instance, a class should exist in the class diagram if the objects of the class exist in the object diagram.

When the inconsistency occurs, we can quickly locate the wrong place based on the feedback information.

Class Existence. The related classes of objects in the object diagrams should exist in the class diagrams since the objects are the instances of the classes.

$$\forall o \in object,\ \exists c \in class,\ o.classid = c.classid.$$

Association Existence. There are links between objects in object diagrams and the associations referred to these links between corresponding classes exist in class diagrams.

$$\begin{aligned}&\forall l \in link,\ \exists o_1, o_2 \in object,\ \exists a \in association,\\ &\quad l.objAid = o_1.objid\ \wedge\ l.objBid = o_2.objid\ \wedge\\ &\quad o_1.classid = a.classAid\ \wedge\ o_2.classid = a.classBid\ \wedge\\ &\quad a.associd = l.associd.\end{aligned}$$

Generalization Satisfaction. If class c_1 is a sub class of class c_2, the domain of c_1 is a subset of the domain of c_2. The set of objects connected to its subclasses should also be disjoint.

$$\forall c_1, c_2 \in class,\ c_1 \prec c_2 \Rightarrow domain(c_1) \subseteq domain(c_2).$$

$$\forall c \in class,\ domain(c) = \bigcup_{c_i \in sub(c)} domain(c_i).$$

$$\forall c \in class,\ \bigcap_{c_i \in sub(c)} domain(c_i) = \emptyset.$$

where function $sub(c)$ gets all the subclasses of class c. Note that the domain of an abstract class is comprised of the domain of the subclasses since there are no instances of an abstract class.

Multiplicity Satisfaction. The number of the instances of an association must satisfy the multiplicity. A set of links should satisfy the multiplicity specifications defined for an association. A minimum and maximum number of instances of target classes must be satisfied using the *lower* and *upper* functions.

$$\begin{aligned}&\forall as \in association,\ lower(multiplicity(as)) \leq\\ &\quad card\{l' \in link(as)\} \leq upper(multiplicity(as)).\end{aligned}$$

Part of the representation of the horizontal consistency rules in Prolog is listed as follows:

```
object_rule(Objid, Classid) :-
  object(Objid, _, Classid),
  class(Classid, _, _),
  write("Object: "), write(Objid), nl,
  write("Class: "), write(Classid).
assoc_exist(Msgid, Sndobjid, Recobjid,
            ClassA, ClassB):-
  link(Msgid, Sndobjid, Recobjid),
  object(Sndobjid, _, ClassA),
  object(Recobjid, _, ClassB),
  association(_, ClassA, ClassB);
  association(_, ClassB, ClassA).
gen_rule(Superid, Subid) :-
  generalization(_, Superid, Subid),
  all_objects_ids(Subid, IDS), write(IDS), nl,
  all_objects_ids(Superid, IDS2), write(IDS2), nl,
  subset(IDS, IDS2).
```

V. Reasoning

We provide an automated tool to transform the models into Prolog. The original models are described in XMI format, generated directly from the UML CASE tools such as StarUML and Astah. Then the models in XMI format are transformed into the Prolog facts automatically. The algorithm of automatic transformation is shown as follows. The code is also available online in [22].

Algorithm 1 Transformation From class diagrams To Prolog

```
Require: The models represent in XMI format.
Ensure: The Prolog Facts of the model.
 1: for each packagedElement ∈ uml:model do
 2:     exact(Class.name).
 3:     exact(Association.name).
 4: end for
 5: for each ownedAttribute ∈ uml:Class do
 6:     if hasAttribute('association') then
 7:         exact(association detail).
 8:         exact(multiplicity).
 9:     else
10:         exact(attribute).
11:     end if
12: end for
13: for each association ∈ association list do
14:     exact(rolename).
15: end for
16: for each ownedOperation ∈ uml:Class do
17:     exact(operation).
18: end for
19: for each generalization ∈ uml:model do
20:     exact(generalization).
21: end for
22: return The Prolog facts of the model.
```

After the model conversion to Prolog facts, the consistency rules along with the converted models are import into SWI-Prolog. The SWI-Prolog checks whether the Prolog-based models satisfy the consistency rules. The models are consistent if the models satisfy all consistency rules. If there are any models that cannot satisfy rules, the SWI-Prolog gives the error messages. Besides, the query interfaces, similar to OCL, are provided to query the relevant information in the models.

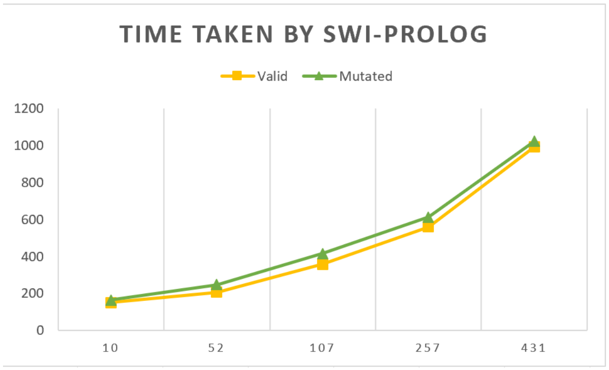
VI. Performance Evaluation

In order to determine the performance of the translation and reasoning tools, we conducted an experiments using UML class diagrams and object diagrams with invariants consisting of 10 - 437 model elements. We use a desktop computer with an Intel(R) Core(TM) i5-7400 CPU processor running at 3.00GHz with 16GB of RAM. The results are shown in Table I.

The performance tests are conducted for both consistent and mutated models. The mutated models contain the randomly introduced inconsistencies such as the same names of models, the direct and indirect cycles in generalizations and

TABLE I
TIME TAKEN BY THE TRANSITION TOOL AND REASONING ENGINES TO PROCESS UML MODELS

Classes	4	7	16	19	32
Model Element	10	52	107	257	431
Translation time (ms)	9.3	39.1	66.7	140.2	321.3
Valid (ms)	150.2	206.9	359.0	561.3	993.5
Mutated (ms)	165.3	248.1	415.3	613.2	1023.1



the multiplicity inconsistencies between the class diagrams and object diagrams. For each test, we measure the time required to translate a model from UML to Prolog and the time required by the SWI-Prolog reasoner to analyze the models. The experimental results show that our approach can find all the inconsistencies mentioned in this paper and the time complexity of Prolog reasoning the consistencies is linear.

VII. CONCLUSION AND FURTHER WORK

In this paper, we present a Prolog-based consistency checking for UML class diagrams and object diagrams. We have implemented an automatic transformation from the models in XMI format to the Prolog facts. Then the SWI-Prolog is used to check whether the facts satisfy the consistency rules. The models are consistent if all consistency rules are satisfied. The reasoning engine will give error information if any inconsistency occurs. Besides, we provide query interfaces, similar with the OCL, to query the relevant information. Our work provides a novel approach to automatically detect the consistency of models and promise the errors will not propagate to the implementation stage.

This study only gives a brief formalization of class diagrams and object diagrams. However, these diagrams are not enough to describe the whole system in real world. We expect to construct a complete UML framework that covers the static structure and dynamic behavior of the systems. Besides, the types of consistency checking in this study are far from enough. More consistency checking rules should be covered to have a confidence for systems.

VIII. ACKNOWLEDGEMENT

This work was partly supported by National Natural Science Foundation of China (Grant No. 61872145) and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No.ZF1213).

REFERENCES

[1] G. Booch, The Unified Modeling Language User Guide, Pearson Education India, 2005.
[2] R. Van Der Straeten, J. Simmonds, and T. Mens, Detecting Inconsistencies between UML Models Using Description Logic, Description Logic, vol. 81, 2003.
[3] J. Woodcock and J. Davies, Using Z: specification, refinement, and proof, Prentice Hall Englewood Cliffs, vol. 39, 1996.
[4] C. Snook and M. Butler, UML-B: Formal modeling and design aided by UML, ACM Transactions on Software Engineering and Methodology(TOSEM), vol. 15(1), pp. 92–122, 2006.
[5] R. S. Bashir, S.P. Lee, S.U.R Khan, V. Chang, and S. Farid, UML models consistency management: Guidelines for software quality manager, International Journal of Information Management, vol. 36(6), pp. 883–899, 2016.
[6] G. Engels, J.M. Küster, R. Heckel, and L. Groenewegen, A methodology for specifying and analyzing consistency of object-oriented behavioral models, ACM SIGSOFT software engineering notes, vol. 26(5), pp. 186–195, 2001.
[7] A. H. Khan and I. Porres, Consistency of UML class, object and statechart diagrams using ontology reasoners, Journal of Visual Languages & Computing, vol. 26, pp. 42–65, 2015.
[8] M. Soeken, R. Wille, and R. Drechsler, Verifying dynamic aspects of UML models, Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6, 2011.
[9] J. Chimiak_Opoka, M. Felderer, and C. Lenz and C. Lange, Querying UML models using OCL and Prolog: A performance study, IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 81–88, 2008.
[10] Z. Khai, A. Nadeem, and G. Lee, A Prolog Based Approach to Consistency Checking of UML Class and Sequence Diagrams, International Conference on Advanced Software Engineering and Its Applications, pp. 85–96, 2011.
[11] M. Egea and V. Rusu, Formal executable semantics for conformance in the MDE framework, Innovations in Systems and Software Engineering, vol. 6(1-2), pp. 73–81, 2010.
[12] H. Störrle, A Prolog-based Approach to Representing and Querying Software Engineering Models, VLL, vol. 274, pp. 71–83, 2007.
[13] P. Krishnan, Consistency checks for UML, 7th Asia-Pacific Proceedings on Software Engineering, pp. 162–169, 2000.
[14] A. Tsiolakis, Consistency analysis of UML class and sequence diagrams based on attributed typed graphs and their transformation, ETAPS 2000 workshop on graph transformation systems, 2000
[15] L. C. Briand, Y. Labiche, L. Osullivan, and M.M. Sowka, Automated impact analysis of UML models, Journal of Systems and Software, vol. 79(3), pp. 339–352, 2006.
[16] D. Torre, Y. Labiche, and M. Genero, UML consistency rules: a systematic mapping study, 18th International Conference on Evaluation and Assessment in Software Engineering, pp. 6, 2014.
[17] D. Torre, Verifying the consistency of UML models, 2016 IEEE International Symposium on Software Reliability Engineering Workshops, pp. 53–54, 2016.
[18] G. Spanoudakis and A. Zisman, Inconsistency management in software engineering: Survey and open research issues, Handbook of software engineering and knowledge engineering, vol. 1, pp. 329–380, 2001.
[19] X. Thirioux, B. Combemale, X. Crégut, and P. Garoche, A framework to formalise the MDE foundations, International Workshop on Towers of Models, pp. 14–30, 2007.
[20] A. Endres and H.D. Rombach, A handbook of software and systems engineering: Empirical observations, laws, and theories, Pearson Education, 2003.
[21] J. Cabot, R. Clarisó, and D. Riera, On the verification of UML/OCL class diagrams using constraint programming, Journal of Systems and Software, vol. 93, pp. 1–23, 2014.
[22] F. Sheng, Transformation from UML to Prolog. "https://github.com/shengfeng/xmi2pl", 2018