I.J. Information Technology and Computer Science, **2018, 9, 47-56**
Published Online September 2018 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2018.09.06

Consistency of UML Design

Iryna Zaretska
V.N. Karazin Kharkiv National University, Kharkiv, 61000, Ukraine
E-mail: zaretskaya@karazin.ua

Oleksandra Kulankhina
Wall Street Systems Software Company, Valbonne, 06560, France
E-mail: oleksandra.kulankhina@gmail.com

Hlib Mykhailenko
ActiveEon Software Company, Valbonne, 06560, France
E-mail: hlib.mykhailenko@gmail.com

Tamara Butenko
V.N. Karazin Kharkiv National University, Kharkiv, 61000, Ukraine
E-mail: tomabut@rambler.ru

Received: 11 November 2017; Accepted: 03 August 2018; Published: 08 September 2018

Abstract—The paper presents a method and tools for consistency checking in UML design of an object-oriented software system. The proposed method uses graph representation of UML diagrams and first-order predicate logic to specify consistency rules mostly on the cross-diagram level. Classification of consistency rules is presented. Two approaches to implementation of consistency checking are discussed and compared.

Index Terms—Software design, object-oriented approach, UML, design model, verification.

I. Introduction

It is well known in software development industry that the earlier faults are detected the less expensive their correcting and the less destructive the wave effect. Supposing that the requirements specification is complete and consistent, the earliest phase of the software life cycle to begin the verification process is logical modeling of the future system. This process requires some language to communicate with users and in between the team members. Here we suppose that UML (Unified Modeling Language [1]) is used to represent logical and physical architecture of a software system. The UML diagrams allow modeling the main aspects of the system such as its static structure, dynamic behavior including events handling, message exchanges, and system state changes. Detecting faults in the UML model of the system i.e. the set of the diagrams reflecting its main characteristics prevents not only improper understanding of specifications but also spreading these faults through further elaboration and coding processes. These faults can be of two kinds. First ones are concerned with UML syntax and semantics and usually are detected by UML CASE tools such as StarUML (http://staruml.sourceforge.net/en/), VisualParadigm (http://www.visual-paradigm.com/), UMLLet (http://www.umlet.com/), Poseidon for UML (http://www.gentleware.com/products.html), IBM Rational Rhapsody (http://www-142.ibm.com/software/products/us/en/ratirhapfami), MagicDraw (http://www.nomagic.com/), ArgoUML (http://argouml.tigris.org/). They are mostly intra-diagram errors. Second ones are concerned with interconnections between diagrams so that the information presented in one diagram does not comply with the same or related information presented in some other diagram. These are mostly structural-to-behavioral or behavioral-to-structural inconsistencies. That kind of faults has nothing to do with UML syntax but is based mostly on rules of object-oriented approach, common sense, and domain understanding. These faults are crucial for the soundness of the future system but, unfortunately, no CASE tools can detect them. So formulating such faults – we call them inconsistencies – and developing methods and tools for their detection is of primary importance. In this paper, we define and classify the wide range of valuable consistency rules and propose two methods of their checking. The decision on changing the model based on the results of such checking remains up to the designer of the system.

Many authors conduct research in this area. Well known are works using description logic as the formal way to represent UML meta-model, the concrete model of the system and consistency rules [3]. Some software tools were presented to support this approach each of which has its own syntax of description for logic propositions and methods of consistency checking (reasoning).

Here we propose another approach using its own model for representing UML diagrams, consistency rules, methods and tools for detecting inconsistencies. As we consider the early phases of the software life cycle and mostly logical modeling, we include into the analysis

only four types of diagrams namely Class diagram, Object diagram, Sequence diagram, and State Machine diagram. One more consideration is that these diagrams have the widest intersection of common UML components (classes, objects, messages, etc.). We use graph representation of the UML model of the system and unified approach of the first order predicate logic to formulate the consistency rules in terms of this representation. Two methods and software tools for checking these rules are proposed and compared. It is supposed that initial UML model is created in some UML CASE tool and is exported into an XMI-file. This file then is parsed into the graph representation and submitted to the software tool (checker) for consistency checking. So the whole process looks like shown in Fig.1.

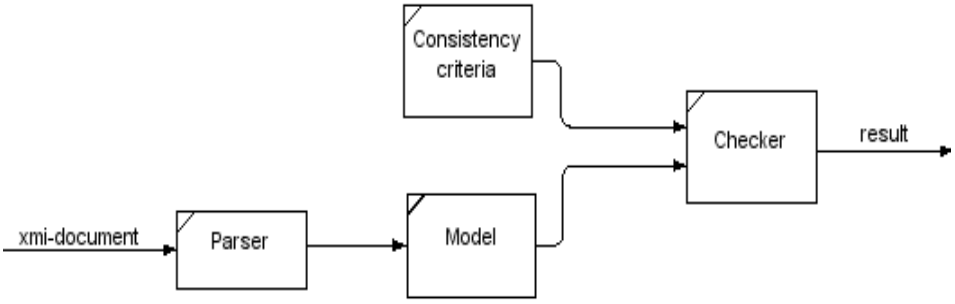

Fig.1. The process of consistency checking.

We developed two essentially different implementations with two different checkers. First uses Java for implementation of all steps of the process. Second is the combination of Java and Prolog to make extensive use of the Prolog's logic machine. The comparative analysis was conducted and the results are presented.

To avoid misunderstanding we use the term "UML design of the system" for the set of its UML models and the term "graph model" for the graph representation of the UML design of the system.

II. Related Work

As the analysis of literature shows the problem of detecting and resolving inconsistencies in UML design models is topical and actively researched. The most complete analysis of approaches is presented in [2]. Some authors restrict their research by only one definite type of diagram while others offer different types of formalisms to describe and detect inconsistencies.

The special classification of inconsistencies in UML models is given in [15].

The closest to our approach is the research carried out in Université de Mons and Vrije Universiteit Brussel, Belgium by a group of authors [3-7] with the description logic (DL) used to solve the problem. The motivation for resorting to DL is natural: it is first-order logic with complete reasoning mechanisms. The meta-model (concepts and roles definitions) form the TBox (Terminological Box) while the concrete model is represented by the ABox (Assertions Box). All the consistency rules should also be written in DL. A number of software tools were developed to support this approach. Among them are RacerPro(http://www.racer-systems.com/), Loom (http://www.isi.edu/isd/LOOM/). The authors of this approach admit that the way to present new consistency rules is neither friendly to the designer nor much expressive.

The latest research conducted by this group in the area involves automated planning and an artificial intelligence technique for automatically generating resolution plans for model inconsistencies [6, 7]. This approach uses a set of 13 structural inconsistency types based on OCL constraints found by authors in the UML meta-model specification.

In [8] translation from UML models to CLP (Constraint Logic Programming) clauses taking advantage of meta-modeling techniques is proposed. CLP is also used to express consistency rules. Then CLP solver used to automatically detect inconsistencies.

The latest research in the area is presented in [11] and it also uses an OCL approach. To carry out the verification of UML consistency models, the following steps were identified in [11]: 1) transformation of UML consistency rules into OCL constraints, 2) generation of a plugin in Papyrus that include the OCL constraints, 3) importation into Papyrus of UML models, 4) execution of the plugin with the OCL constraints against the imported UML models.

The authors of [12] classified existing proposed techniques based on the parameters identified from the research literature. They performed a qualitative comparison of consistency management techniques in order to identify current research trends, challenges and research gaps in this field of study. Based on the results, they concluded that researchers have not provided sufficient attention to exploring inter-model and semantic consistency problems.

Some methods of verifying UML/OCL models are presented in [13], [14].

So the problem of detecting structural-to-behavioral or behavioral-to-structural inconsistencies based not on OCL rules but on principles of object-oriented design remains open.

III. Graph Model of UML Design

The motivation for using the graph model is natural as diagrams in fact are graphs and moreover those four diagrams considered in this chapter have common or closely related vertices and edges. This fact is extensively used in formulating the consistency rules. In fact, graph representation simplifies the description of diagrams comparing to their formal specification [1] but is sufficient for verification purposes. For a class diagram, the corresponding graph's vertices are classes and edges are connections between them, which are association, dependency, generalization, and interface implementation. The information about generalization sets is stored separately to simplify search algorithms. For an object diagram, the corresponding graph's vertices are objects and edges are links between them. For a sequence diagram, the vertices are objects or classes and edges are messages between them. For a state machine (or state chart) diagram the vertices are states and edges are transitions between them. Each type of vertex and each type of edge stores information needed to check intra- and cross-diagram inconsistencies. For example, an association in a class diagram as an edge of the corresponding graph keeps the name of the association, roles and multiplicities of its participants, etc. Here is the formal representation of our model consisting of graphs of four types for Class, Object, Sequence, and State Machine diagrams respectively:

$$D = \{\{D_{cl}\} \cup \{D_{ob}\} \cup \{D_{seq}\} \cup \{D_{st}\}\} \tag{1}$$

Each of these graphs consists of two sets: V stands for vertices and E stands for edges. Their description is given below.*

$$\begin{gathered}
D_{cl} = \{V_{cl}, E_{cl}\} \\
V_{cl} = \{v : v = \\
(name, isAbstract[, ATTR, MTHD, STRT, visibility])\} \\
ATTR = \{attr : attr = \\
(name, domain, scope[, visibility, multiplicity])\} \\
MTHD = \{mthd : mthd = \\
(mthdSgn, scope, visibility)\} \\
scope = instance \mid classifier \\
visibility = public \mid private \mid protected \mid package \\
mthdSgn = (name[, PARAMS, returnDomain]) \\
PARAMS = \{param : \\
param = (num[, name], domain)\} \\
STRT = \{stereotype : stereotype = (name)\} \\
E_{cl} = \{e : e = (v_s, v_e, type[, info]); \\
v_s, v_e \in V_{cl}, type = gen \mid ass \mid dep \mid impl\} \\
info = ([name, r_s, r_e, m_s, m_e, aggr_s, \\
aggr_e, navig_s, navig_e]) \\
D_{ob} = \{V_{ob}, E_{link}\}
\end{gathered} \tag{2}$$

$$\begin{gathered}
V_{ob} = \{v : v = \\
(name, clName[, ATTRVAL, STRT)\} \\
ATTRVAL = \\
\{attrval : attrval = (name, value)\} \\
E_{link} = \{e : e = (v_s, v_e, name); v_s, v_e \in V_{ob}\} \\
D_{seq} = \{V_{cl} \cup V_{ob}, E_{msg}\} \\
E_{msg} = \{e : e = \\
(v_s, v_e, msgCall); v_s, v_e \in V_{cl} \cup V_{ob}\} \\
msgCall = ([guard,] seqnum, mthdCall) \\
mthdCall = (name, ARGS[, returnValue]) \\
ARGS = \{armnt : armnt = (num, value)\} \\
D_{st} = \{V_{st}, E_{tr}, className\} \\
V_{st} = \{v : v = (name[, entry, do, exit]); \\
entry, do, exit \in mthdCall\} \\
status = start \mid final \\
E_{tr} = \{e : e = (v_s, v_e, trCall); v_s, v_e \in V_{st}\} \\
trCall = ([guard,] mthdCall).
\end{gathered}$$

Detailed examples of this model are presented in [9, 10]. In fact (2) defines the meta-model and any concrete design is represented as a graph model compliant to it.

IV. Consistency Rules

In general, the consistency rules state that all the structural elements of the system (mainly classes and objects) presented in the behavioral diagrams like sequence and state machine ones should be presented in the structural diagrams like class and object ones with proper types of associations and links, visibility and navigation types, multiplicities, etc. And as the messages, in fact, are the objects' methods calls and state transitions mean methods invocations, these behavioral elements should have their proper presentation in structural diagrams. And all these rules should take into account the basics of object-oriented design. For example, checking the presence of some method in a class may mean checking this method along the hierarchy path up to the base class or interface with the generalization or implementation types of connecting edges. Exactly for this purpose, the following notation is introduced:

$$\begin{gathered}
implGenPath(v) == v_1 ... v_n : ((v \in V_{cl} \wedge v_1 = v) \vee \\
(v \in V_{ob} \wedge (\exists cl \in V_{cl} : clName(v) = name(cl)) \wedge \\
v_1 = cl)) \wedge (\forall i = 1, n-1) \\
(\exists e \in E_{Cl} : (type(e) = gen \vee type(e) = impl) \wedge \\
v_s(e) = v_i \wedge v_e(e) = v_{i+1}
\end{gathered} \tag{3}$$

In this section the main consistency rules are classified, their description and unified presentation in terms of the model (1, 2) by normal logic formulae is given. We present here not all but the most valuable rules.

* Elements in [] are optional.

4.1 Class diagrams vs. Sequence diagrams

This section specifies structural-to-behavior consistency rules, which cover class and sequence diagrams in the UML design of the system.

1. If an instance of class A sends the message to an instance of class B in the Sequence diagram, the class B should be visible for the class A in the Class diagram with proper visibility modifier.

$$\begin{gathered}(\forall e \in E_{msg} : v_e(e) \in V_{cl}) \\ (\exists v \in implGenPath(v_e(e))) \\ (\exists mthd \in MTHD(v(e))) \\ (msgCall(e) \approx mthdSgn(mthd)) \\ ((visibility(mthd) = "public") \vee \\ ((visibility(mthd) = "protected") \wedge \\ ((v_s(e) \in V_{cl} \wedge v_e(e) \in implGenPath(v_s(e)) \vee \\ ((v_s(e) \in V_{ob}) \wedge \\ (\exists cl \in V_{cl} : name(cl) = clName(v_s(e))) \wedge \\ (v_e(e) \in implGenPath(cl)) \wedge \\ (\forall e \in E_{msg} : v_e(e) \in V_{ob})(\exists cl \in V_{cl}) \\ (\exists v \in implGenPath(v_e(e))) \\ (\exists mthd \in MTHD(v(e))) : msgCall(e) \approx \\ mthdSgn(mthd))((visibility(mthd) = "public") \vee \\ ((visibility(mthd) = "protected") \wedge \\ ((v_s(e) \in V_{cl} \wedge v_e(e) \in implGenPath(v_s(e)) \vee \\ ((v_s(e) \in V_{ob}) \wedge \\ (\exists cl \in V_{cl} : name(cl) = clName(v_s(e))) \wedge \\ (v_e(e) \in implGenPath(cl)) \end{gathered} \tag{4}$$

2. If an instance of class A sends the message to an instance of class B in the Sequence diagram there should be the corresponding method in the class B.

$$\begin{gathered}(\forall e \in E_{msg} : v_e(e) \in V_{cl}) \\ (\exists v \in implGenPath(v_e(e))) \\ (\exists mthd \in MTHD(v)) \\ (msgCall(e) \approx mthdSgn(mthd)) \wedge \\ (\forall e \in E_{msg} : v_e(e) \in V_{ob}) \\ (\exists cl \in V_{cl} : name(cl) = clName(v_e(e))) \\ (\exists v \in implGenPath(v_e(e))) \\ (\exists mthd \in MTHD(v)) \\ (msgCall(e) \approx mthdSgn(mthd)) \end{gathered} \tag{5}$$

3. If an instance of class A sends the message to the class B in the Sequence diagram the invoked method should be declared as static.

$$\begin{gathered}(\forall e \in E_{msg} : (\exists v \in implGenPath(v_e(e))) \\ (\exists mthd \in MTHD(v(e))) \\ (msgCall(e) \approx mthdSgn(mthd) \wedge \\ scope(mthd) = "classifier") \end{gathered} \tag{6}$$

4. In the Sequence diagrams there should not be present objects of the class with the "utility" stereotype specified in the Class diagram.

$$\begin{gathered}(\forall o \in E_{ob} \in D_{seq}) \\ (\exists c \in V_{Cl} : name(c) = clName(o)) \\ (\forall str \in STRT(c)) \\ (name(str) \neq "utility") \end{gathered} \tag{7}$$

5. If some class has the multiplicity of its association end equal to 1 in the Class diagram then the message can be sent only to one instance of this class in the Sequence diagram.

$$\begin{gathered}(\forall cl \in E_{Cl}) \\ (\forall e \in E_{Cl} : type(e) = ass \wedge \\ multiplicity_e = 1) \\ (\exists e1 \in E_{msg} : v_e(e1) \in V_{ob} : name(cl) = \\ clName(v_e(e1))) \\ (\exists mthd \in MTHD(cl)) \\ (msgCall(e1) \approx mthdSgn(mthd)) \wedge \\ (\neg((\exists e2 \in E_{msg} : v_e(e2) \in V_{ob} : \\ name(cl) = clName(v_e(e2))) \\ (\exists mthd \in MTHD(cl)) \\ (msgCall(e2) \approx mthdSgn(mthd)))) \end{gathered} \tag{8}$$

6. Navigation parameters of the associations defined in the Class diagram should comply with Navigation parameters used in the Sequence diagram.

$$\begin{gathered}(\forall e \in E_{Cl} : navigation_e = true) \\ (\neg(\exists e \in E_{msg} : v_s(e) \in V_{Cl}) \\ (\exists mthd \in MTHD(v_s(e))) \\ (msgCall(e) \approx mthdSgn(mthd))) \wedge \\ (\forall e \in E_{Cl} : navigation_s = true) \\ (\neg(\exists e \in E_{msg} : v_e(e) \in V_{Cl}) \\ (\exists mthd \in MTHD(v_e(e))) \\ (msgCall(e) \approx mthdSgn(mthd))) \end{gathered} \tag{9}$$

4.2 Class diagrams vs. Object diagrams

This section specifies structural consistency rules, which cover class and object diagrams in the UML design of the system. We use here

$$\begin{gathered}LINKS(v_{ob}, v_{cl}) = \\ \{e \in E_{link} : (v_s(e) = v_{ob} \wedge \\ (\exists v \in children(v_{cl}) : name(v) = \\ clName(v_e(e))) \vee (v_e(e) = v_{ob} \wedge \\ (\exists v \in children(v_{cl}) : name(v) = \\ clName(v_s(e)))\} \end{gathered} \tag{10}$$

to denote the set of edges connecting some vertex-object with vertices-instances of definite class or its generalizations and

$$children(v) = \{ch \in V_{cl} : \exists path = implGenPath(ch) : v \in path\} \quad (11)$$

to denote all possible implementations or generalizations of the class.

1. In case of composite aggregation the "part"-object can belong to only one "whole"-object.

$$\begin{gathered}(\forall e \in E_{cl} : type(e) = ass \wedge \\ aggr_s(e) = composite) \\ (\forall part\_cl \in children(v_e(e))) \\ (\forall part\_ob \in V_{ob} : clName(part\_ob) = \\ name(part\_cl)) \\ (|LINKS(part\_ob, v_s(e)| \leq 1) \wedge \\ (\forall e \in E_{cl} : type(e) = ass \wedge \\ aggr_e(e) = composite) \\ (\forall part\_cl \in children(v_s(e))) \\ (\forall part\_ob \in V_{ob} : clName(part\_ob) = \\ name(part\_cl)) \\ (|LINKS(part\_ob, v_e(e)| \leq 1)\end{gathered} \quad (12)$$

2. In the Object diagram, there should not be present objects of the class with the "utility" stereotype specified in the Class diagram.

$$\begin{gathered}(\forall o \in E_{ob}) \\ (\exists c \in V_{Cl} : name(c) = clName(o)) \\ (\forall str \in STRT(c)) \\ (name(str) \neq "utility")\end{gathered} \quad (13)$$

3. The objects in the Object diagram should have relationship only if:

- there is a relationship between corresponding classes or any of their parent classes in the Class diagram and this relationship is association;
- one of the corresponding classes or any its parent has an attribute with the type of other class or of any its parent in the Class diagram.

$$\begin{gathered}(\forall l \in E_{link}) \\ (\exists cl\_s \in E_{cl} : name(cl\_s) = clName(v_s(l))) \\ (\exists cl\_e \in E_{cl} : name(cl\_e) = \\ clName(v_e(l)))(\exists v \in genPATH(cl\_s)) \\ (\exists u \in genPATH(cl\_e)) \\ ((\exists e \in E_{cl} : type(e) = ass \wedge \\ (v_s(e) = v \wedge v_e(e) = u \wedge \\ name(v_s(l)) \equiv r_s(e) \wedge name(v_e(l)) \equiv r_e(e))) \vee \\ (v_e(e) = v \wedge v_s(e) = u \wedge \\ name(v_s(l)) \equiv r_e(e) \wedge \\ name(v_e(l)) \equiv r_s(e))))) \vee \\ (\exists at1 \in ATTR(v) : domain(at1) = name(u) \wedge \\ name(at1) \equiv name(v_e(l))) \vee \\ (\exists at2 \in ATTR(u) : domain(at2) = name(v) \wedge \\ name(at2) \equiv name(v_s(l)))\end{gathered} \quad (14)$$

4. The value of the object attribute in the Object diagram should not contradict to its type specified in the Class diagram.

$$\begin{gathered}((\forall ob \in V_{ob})(\forall attrval \in ATTRVAL(ob)) \\ (\exists cl \in V_{Cl} : ClName(ob) = name(cl)) \wedge \\ \big((\exists v \in genPATH(cl))((\exists attr \in ATTR(v)) \\ (name(attr) \neq name(attrval) \vee \\ domain(attrval) \in domain(attr))\end{gathered} \quad (15)$$

5. For each attribute of the object in the Object diagram:

- the corresponding class should have the attribute with the same name;
- there should be association between corresponding class or any its parent and other class, and association role should not contradict to the attribute name.

$$\begin{gathered}(\forall ob \in V_{ob})(\forall attrval \in ATTRVAL(ob)) \\ (\exists cl \in V_{Cl} : ClName(ob) = name(cl)) \\ \big((\exists v \in genPATH(cl)) \\ (\exists attr \in ATTR(v) : name(attrval) = \\ name(attr)) \vee \\ (\exists e \in E_{Cl} : type(e) = ass) \\ (v_s(e) = v \wedge (r_e(e) = name(attrval) \vee \\ (v_e(e) = v \wedge (r_s(e) = name(attrval)))\end{gathered} \quad (16)$$

6. In an association relationship, the number of instances associated with corresponding instances in the Object diagram should not contradict to multiplicities of the association ends in the Class diagram.

$$\begin{gathered}(\forall e \in E_{cl} : v_s(e) \in V_{Cl} : type(e) = ass) \\ (\forall o \in V_{ob} : clName(o) = name(v_s(e))) \\ (|\exists o| = multiplicity_s(e)) \vee \\ (\forall e \in E_{cl} : v_e(e) \in V_{Cl} : type(e) = ass) \\ (\forall o \in V_{ob} : clName(o) = name(v_e(e))) \\ (|\exists o| = multiplicity_s(e))\end{gathered} \tag{17}$$

7. An object of the class with the "implementationClass" stereotype should not be an instance of more than one class.

$$\begin{gathered}(\forall o \in V_{Ob}) \\ (\forall cl1 \in V_{Cl} : name(cl1) = clName(o)) \\ (\neg(\exists str \in STRT(cl1)) \\ (name(str) = "implementationClass")) \vee \\ (\neg((\exists cl2 \in V_{Cl} : name(cl2) = clName(o))))\end{gathered} \tag{18}$$

4.3 Class diagrams vs. State Machine diagrams

This section specifies structural-to-behavior consistency rules, which cover class and state machine diagrams in the UML design of the system.

1. As transition from one state of the class A to another one in the State Machine diagram takes place by the class A method invocation there should be such method in the class A in the Class diagram.

$$\begin{gathered}(\forall e \in E_{tr}) A \\ (\exists cl \in V_{cl} : name(cl) = clName(D_{st})) \\ (\exists v \in implGenPath(v_e(e))) \\ (\exists mthd \in MTHD(v)) \\ (trCall(e) \approx mthdSgn(mthd))\end{gathered} \tag{19}$$

4.4 Sequence diagrams vs. State Machine diagrams

This section specifies behavior consistency rules, which cover sequence and state machine diagrams in the UML design of the system.

1. The order of the messages sent in the Sequence diagram should not contradict to the order of the corresponding transitions from one state of the class to another one in the State Machine diagram.

$$\begin{gathered}(\forall e_{1tr} \in E_{tr}) \\ ((\exists e_{1msg} \in E_{msg} : mthdCall(trCall(e_{1tr})) = \\ mthdCall(msgCall(e_{1msg}))) \\ (num1 \in seqnum(msgCall(e_{1msg})) \wedge \\ (\forall e_{2tr} \in E_{tr} : v_s(e_{2tr}) = v_e(e_{1tr}))\end{gathered} \tag{20}$$

$$\begin{gathered}(\exists e_{2msg} \in E_{msg} : mthdCall(trCall(e_{2tr})) = \\ mthdCall(msgCall(e_{2msg}))) \\ (num2 \in seqnum(msgCall(e_{2msg})) \\ (num1 < num2))\end{gathered}$$

4.5 Class diagrams (intra-diagram rules)

This section specifies structural consistency rules, which cover only class diagrams in the UML design of the system.

1. In case of composite aggregation the "part"-object can belong to only one "whole"-object at a time.

$$\begin{gathered}(\forall e \in E_{Cl} : type(e) = ass \wedge \\ aggr_s(e) = composite) \\ (multiplicity_s(e) \le 1) \wedge \\ (\forall e \in E_{Cl} : type(e) = ass \wedge \\ aggr_e(e) = composite) \\ (multiplicity_e(e) \le 1)\end{gathered} \tag{21}$$

2. The class with the "utility" stereotype should have only static members.

$$\begin{gathered}(\forall v \in V_{Cl} : \exists strt \in STRT(v): \\ name(strt) = "utility") \\ ((\forall mthd \in MTHD(v)) \\ (scope(mthd) = "classifier")) \wedge \\ ((\forall attr \in ATTR(v)) \\ (scope(attr) = "classifier"))\end{gathered} \tag{22}$$

3. The class stereotypes should be compatible (for instance "enum" and "interface" stereotypes are incompatible).

$$\begin{gathered}(\forall v \in V_{Cl})(\neg(\exists st1 \in STRT : \\ name(st1) = "enum")) \vee \\ (\neg(\exists st2 \in STRT : \\ name(st2) = "interface"))\end{gathered} \tag{23}$$

4. At least one end of the association should have true value for the Navigability parameter.

$$\begin{gathered}(\forall e \in E_{Cl} : type(e) = ass) \\ (navig_s(e) \neq false \vee navig_e(e) \neq false)\end{gathered} \tag{24}$$

5. Only the binary association can be of the aggregation or composition type.

$$
\begin{array}{c}
(\forall e \in E_{Cl} : type(e) = ass \wedge \\
aggr_s(e) = composite) \\
((\exists cl1(v_s(e)))(\exists cl2(v_e(e))) \wedge \\
(\neg(\exists cl3(v_e(e))))) \vee \\
(\forall e \in E_{Cl} : type(e) = ass \wedge \\
aggrv_e(e) = composite) \\
((\exists cl1(v_e(e)))(\exists cl2(v_s(e))) \wedge \\
(\neg(\exists cl3(v_s(e)))))
\end{array} \quad (25)
$$

4.6 State Machine diagram (intra-diagram rules)

This section specifies behavioral consistency rules, which cover only state machine diagrams in the UML design of the system.

1. Any state should be reachable from the start state.

$$
\begin{array}{l}
(\forall v \in V_{st}) \\
(\exists v_1, \ldots, v_k = v : ((start(v_1) = true) \wedge \\
((\forall i = 1, k-1) \\
(\exists e \in E_{tr} : (v_s(e) = v_i \wedge v_e(e) = v_{i+1})))
\end{array} \quad (26)
$$

2. The final state should be always reachable.

$$
\begin{array}{l}
(\forall v \in V_{st}) \\
(\exists v_k = v, \ldots, v_n : ((final(v_n) = true) \wedge \\
((\forall i = k, n-1) \\
(\exists e \in E_{tr} : (v_s(e) = v_i \wedge v_e(e) = v_{i+1})))
\end{array} \quad (27)
$$

V. Implementation of Consistency Rules Checking

Two approaches to consistency rules checking were implemented. Both suppose the input to be an XMI-file containing UML design of the system.

The first approach uses Java classes to implement all components of the verification process: graph meta-model of UML design, graph model of the UML design under checking, and consistency rules (criteria). The special converter parses XMI-file into Java classes of the concrete graph model. Then Java checker verifies it according to the criteria. The workflow of this process in the IDEF0 notation is shown in Fig. 2. The class diagram for consistency criteria structure is shown in Fig. 3.

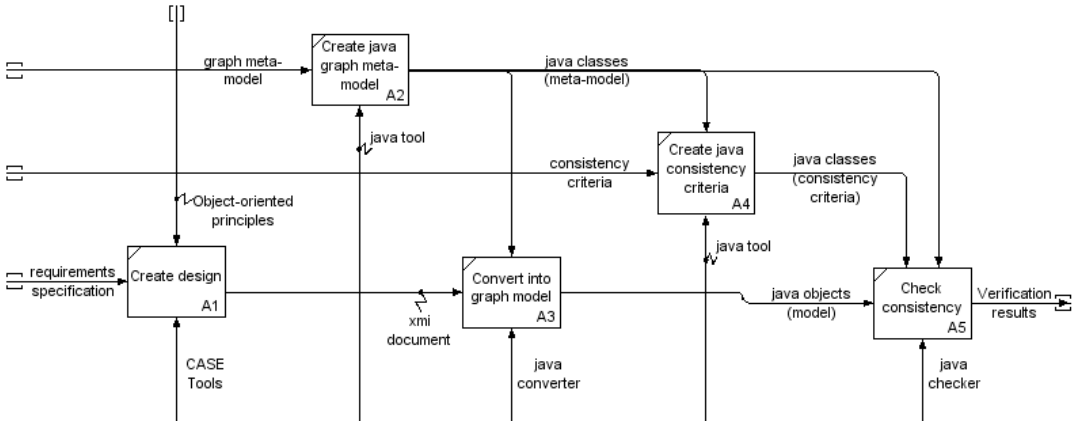

Fig.2. The process of consistency checking with java checker.

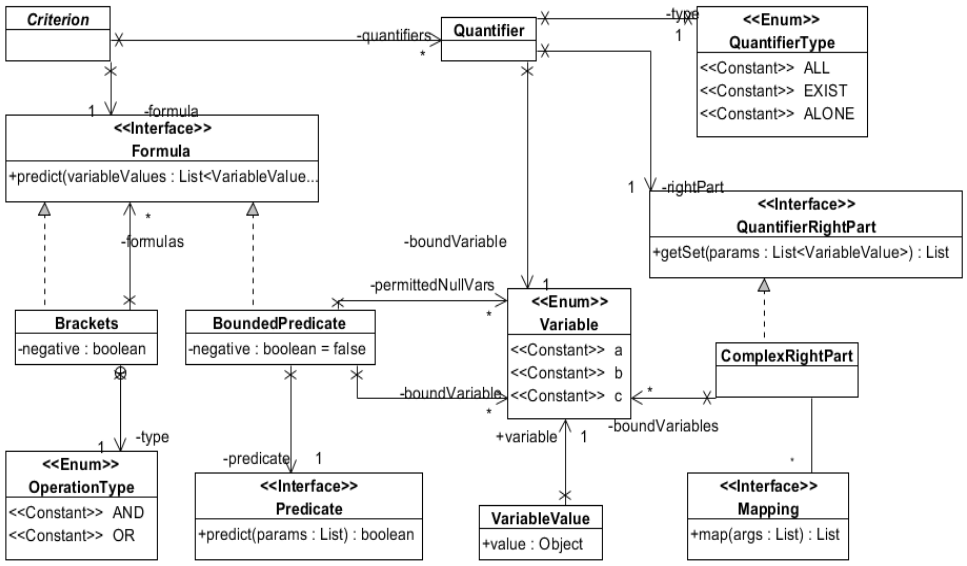

Fig.3. Class diagram for consistency criteria structure.

The second approach delegates checking responsibilities to the Prolog reasoning engine and uses Java as the framework for preparatory tasks such as converting initial XMI-file into the text file with Prolog facts. The JPI (Java-Prolog Interface) is used as a bridge between Java and SWI-Prolog implementation. In this approach the graph meta-model, as well as consistency rules, are represented as Prolog facts and rules respectively in text files in advance. The workflow of this process in the IDEF0 notation is shown in Fig. 4.

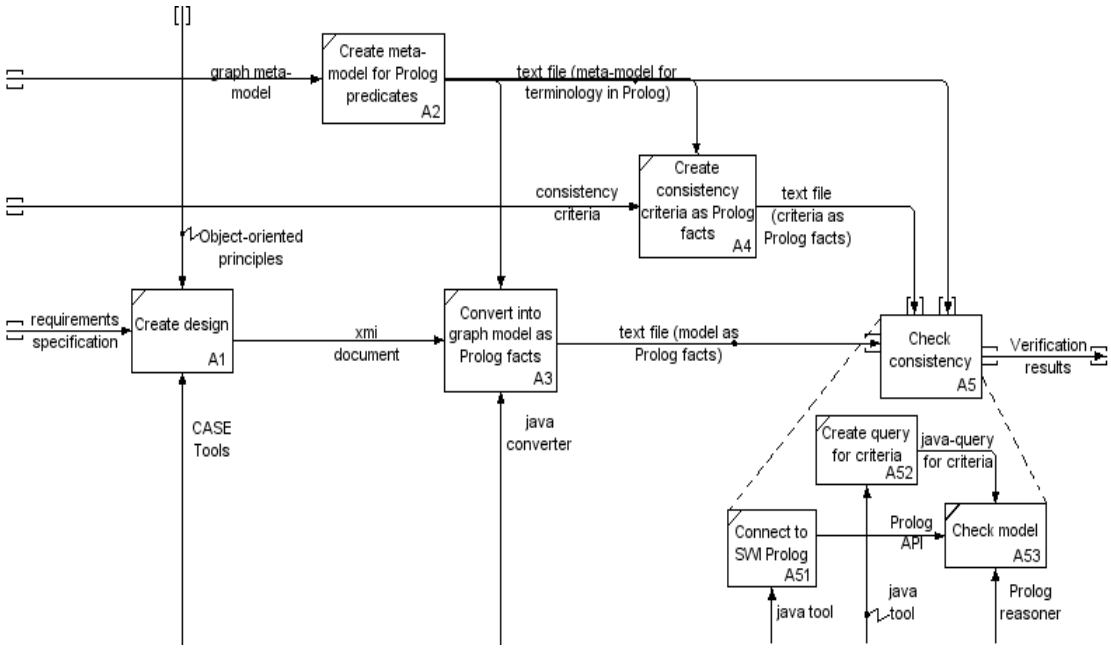

Fig.4. The process of consistency checking with prolog reasoning engine.

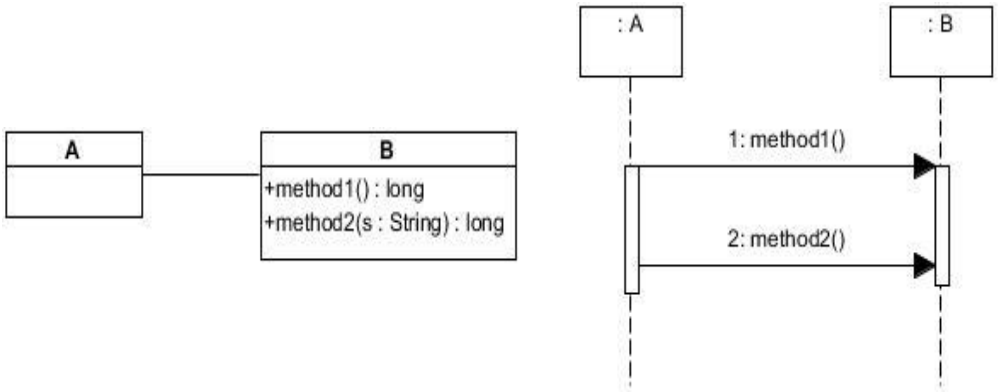

Fig.5. Fragment of the UML design.

For the small fragment of UML design given in Fig. 5 the XMI-file is converted into the following Prolog facts:

```
vclass(a, false, public)
vclass(b, false, public)
eclass(eclid1, a, b, as)
mthd(mthdid1, b, method1, public , instance, long)
mthd(mthdid2, b, method2, public , instance, long)
param(mthdid2, 0, s, string)
vseq(vseqid1, undefined , a, object)
vseq(vseqid2, undefined, b, object)
eseq(eseqid1, vseqid1, vseqid2)
eseq(eseqid2, vseqid1, vseqid2)
msgCall(msgcallid1, eseqid1, 1, method1, undefined)
msgCall(msgcallid2, eseqid2, 2, method2, undefined).
```

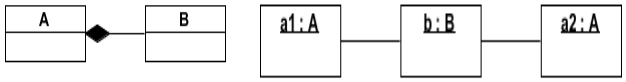

Fig.6. Fragment of the UML design.

To check the consistency criteria (12) for the fragment given in Fig.6 the following Prolog facts and rules are used:

```
vclass(a, false, public)
vclass(b, false, public ).
eclass(eclassSurrogateID0,a, b, as)
```

info(eclassSurrogateID0, undefined, undefined, undefined, undefined, undefined, composite, none, undefined, undefined)
vobject(a1, a)
vobject(b, b)
vobject(a2, a)
link(a1,b, undefined)
link(a2,b, undefined)
link(X,Y) :- link(Y,X)
p1(B):-vclass(CA,_,_), vclass(CB,_,_), composition(CA,CB), vobject(A,CA), vobject(B,CB), vobject(C,CA), link(A,B,_), link(C,B,_).

VI. Evaluation Results

As the part of the research, we checked inconsistencies in a number of real projects of different scope using both developed tools.

As results show the second approach proved to be more efficient as it uses verified reasoning mechanisms and text files are much simpler than corresponding Java objects. The only disadvantage of the second way is the necessity to install Prolog software.

The analysis of results shows that the most common inconsistencies found by the developed tools are of three types: a message call for a nonexistent method, sending a message to an instance of a nonexistent class and a message call for a method with an unaccepted visibility modifier.

The results have been compared with experts' evaluation of these projects. For small to middle-size projects, an expert can find less than a half of inconsistencies found by the proposed tools while for bigger projects (with more than 100 classes and more 200 messages) an expert can find only 10%-15% of inconsistencies found by the proposed tools.

So the developed tools could assist a designer to avoid not only misprints but also an inaccurate assignment of responsibilities between classes.

VII. Conclusions

The paper offers a general method for checking consistency in UML design of an object-oriented system. It uses the unified model with graph representation of the design components and formulae of the first order predicate logic to represent consistency criteria. The main cross-diagrams and intra-diagrams criteria are presented and classified.

Two approaches to consistency checking implementation are offered: Java framework with its own reasoning engine and Java framework with the bridge to Prolog reasoning engine. The second approach proved to be better in several ways as it uses verified software in its most critical part and simple text files for facts and rules representation.

Both approaches can be used to evaluate the quality of a design and make recommendations on its improvement due to the better use of main principles of the object-oriented design.

References

[1] Object Management Group: UML 2.0 Superstructure Specification (2005), http://www.uml.org/

[2] Francisco J. Lucas, Fernando Molina, Ambrosio Toval: A systematic review of UML model consistency management. In: In Information and Software Technology, Vol. 51, p 1631—1645, (2009)

[3] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers: Using Description Logic to Maintain Consistency between UML Models. In: Proc. UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings, 2003, pp. 326--340., (2003)

[4] Ragnhild Van Der Straeten: Description of UML Model Inconsistencies. Vrije Universiteit Brussel, Department of Computer Science, SOFT-TR-2011.01.15 (2011)

[5] Jorge Pinna Puissant, Tom Mens, Ragnhild Van Der Straeten: Comparing Automated Planning Approaches for Model Inconsistency Resolution. Technical report, University of Mons, 2011-04-10, Travail sans promoteur/Rapport de recherche (2011)

[6] Jorge Pinna Puissant, Tom Mens: Resolving Inconsistencies in Model-Driven Engineering using Automated Planning. In: Seminar on Advanced Tools & Techniques for Software Evolution (SATToSE), Koblenz, Germany, 2012 (2012)

[7] Jorge Pinna Puissant, Ragnhild Van Der Straeten, Tom Mens: Badger: A Regression Planner to Resolve Design Model Inconsistencies. In: Modelling Foundations and Applications, Lecture Notes in Computer Science, Volume 7349 , pp 146--161 (2012)

[8] H. Malgouyres, G. Motet: A UML model consistency verification approach based on meta-modeling formalization. In: Proceedings of the 2006 ACM symposium on Applied computing, pp 1804--1809 (2006)

[9] Iryna Zaretska, Roman Kovalenko, Oleksandra Kulankhina, and Hlib Mykhailenko: Checking inconsistencies in UML design. In: http://ceur-ws.org/Vol-848/ICTERI-2012-CEUR-WS-paper-4-p-33-43.pdf

[10] Iryna Zaretska, Oleksandra Kulankhina, and Hlib Mykhailenko: Cross-Diagram UML Design Verification. In: V. Ermolayev et. al. (eds.) ICT in Education, Research and Industrial Applications. CCIS, Vol. 347, pp. 165--176. Springer-Verlag, Berlin Heidelberg (2013)

[11] Damiano Torre: Verifying the Consistency of UML Models. In: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE *Xplore* (2016)

[12] Raja Sehrab Bashir, Sai Peck Lee, Saif Ur Rehman Khan, Victor Chang, Shahid Farid: UML models consistency management. In: International Journal of Information Management: The Journal for Information Professionals, Volume 36, Issue 6, December 2016 pp 883--899 (2016)

[13] N. Przigoda, M. Soeken, R. Wille and R. Drechsler: Verifying the structure and behavoir in UML/OCL models using satisfiability solvers. In: IET Cyber-Physical Systems: Theory and Applications, Vol. 1, Issue 1, pp 49--59 (2016).

[14] N. Przigoda, J.G. Filho, Ph. Niemann, R. Wille and R. Drechsler: Frame conditions in symbolic representation of UML/ OCL models. In: 2016 ACM/ IEEE International Conference on Formal Methods and System Design, pp

178--185 (2016)
[15] D. Allaki, M. Dahchour, A. Nouaary: A new taxonomy of inconsistencies in UML models: with their detection methods for better MDE. In: International Journal of Computer Science and Applications, Vol. 12, No. 1, pp 48--65 (2015)

Authors' Profiles

Iryna Zaretska obtained her PhD degree in Mathematics from the V. N. Karazin Kharkiv National University, Ukraine, in 1990.

Currently, she works as a professor of V. N. Karazin Kharkiv National University, School of Mathematics and Informatics, Department of Theoretical and Applied Computer Science.

Her research interests include software design, formal verification, and model-driven engineering.

Oleksandra Kulankhina obtained her PhD degree in Computer Science from the University of Nice-Sophia Antipolis, Nice, France, in 2016.

Currently, she works as a Research and Developement Software Engineer at Wall Street Systems, Valbonne, France.

Her research interests include distributed systems, formal verification, and model-driven engineering.

Hlib Mykhailenko obtained his PhD degree in computer science from the University of Nice-Sophia Antipolis, Nice, France, in 2017.

Currently, he works as a Research and Developement Software Engineer at ActiveEon, Valbonne, France. His research interests include large-scale distributed programming, graph partitioning algorithms, and functional programming.

Tamara Butenko obtained her post-diploma education in mechanical engineering from the Kharkiv Institute of Mechanical Engineering Problems, Ukraine, in 2008.

Currently, she works as a senior teaching staff member at V. N. Karazin Kharkiv National University, School of Physics, Department of Higher Mathematics.

How to cite this paper: Iryna Zaretska, Oleksandra Kulankhina, Hlib Mykhailenko, Tamara Butenko, "Consistency of UML Design", International Journal of Information Technology and Computer Science(IJITCS), Vol.10, No.9, pp.47-56, 2018. DOI: 10.5815/ijitcs.2018.09.06

