

A Metamodel Based Approach for Inconsistency Detection

Mohammad Ariyan¹, Jafar Habibi² and Ali Kamandi³

¹ Sharif University of Technology, Tehran, Iran
ariyan@ce.sharif.edu

² Sharif University of Technology, Tehran, Iran
jhabibi@sharif.edu

³ Sharif University of Technology, Tehran, Iran
kamandi@ce.sharif.edu

Abstract. One of the important issues in software modeling languages is consistency management inside models. Although UML as a standard for software modeling has been very successful compared to its predecessors, inconsistency is one of its unresolved problems. In recent years, much work has been done to detect inconsistencies in models of a language (e.g. UML), but limited research has been done to detect inconsistencies in the metamodel level. This paper presents a method for discovering possible inconsistencies at the modeling language design level. The precise semantics of the metamodel along with a target (possibly inconsistent) model are the inputs to this method. We define a translation of the input into first order predicate logic and logical reasoning is used to detect possible inconsistencies. This method will help modelers prevent modeling inconsistencies.

Metamodel, UML, Model, Inconsistency, First order predicate logic.

1 Introduction

The Unified Modeling Language (UML) [23] is a standard language; it is used as a visual tool for designing software systems. However, the ambiguity of UML along with its support of modeling with different viewpoints pose a great risk of inconsistency. Many classifications of UML inconsistencies exist in the literature today. Several proposals are also made to clear the existing ambiguity and to provide methods for detecting possible inconsistencies.

The notion of consistency has been investigated in different domains and at various levels. The Webster's dictionary definition of inconsistency is: 'the relation between propositions that cannot be true at the same time, or the lack of harmonious uniformity among parts'. In the context of UML various definitions for inconsistency have been provided. In [9] inconsistency has been defined as 'a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not

jointly satisfiable'. We believe that this definition would be more precise if it was completed at the end with '... which are not jointly satisfiable based on rules and constraints provided in the Metamodel'.

The first motivation for model consistency is correctness. Completeness is the second motivation. Usually, inconsistency is caused by design problems or misuse of UML. When these problems are discovered early in the design process, it is easier and less costly to fix them. The third motivation is implementability, which usually involves translating a UML model into a programming language, which is a precise and unambiguous notation.

This paper presents a method for discovering possible inconsistencies at the modeling language design level. Metamodel is the main input of this method. Metamodel semantics along with a target model (which we are trying to determine if it is consistent) are translated into first order predicate logic and logical reasoning is then used to detect possible inconsistencies.

The novelty of our approach is that:

- The precise semantics of language design will be considered. In this way, models will be checked against every rule defined in the metamodel and not a selected subset of those rules (other methods select a restricted subset of inconsistency types based on designers own experience and focus only on algorithms for managing those inconsistencies).
- This method, unlike others, is not bound to a specific type of model and consistency of any kind of model can be analyzed with this method as long as the metamodel specification for that kind is available.

UML specifications introduce two main structures that can be used for consistency analysis: metamodel diagrams and well-formedness rules. Metamodel diagrams are schemas that define rules and constructs for model creation. For example, 'an association has at least two association ends'. Figure 1 shows a simplified subset of the UML class diagram definition.

A model may not be consistent even if it conforms to the metamodel. Metamodel diagrams are not expressive enough to provide all the aspects of UML specifications. There is a need to describe additional constraints about metamodel objects. OCL has been developed to fill this gap. OCL is a higher order logic language for expressing well-formedness rules [25]. For example the following OCL expression determines that association ends must have a unique name within the association:

Context Association inv:

$self.allConnections \rightarrow forAll(r_1, r_2 | r_1.name = r_2.name \text{ implies } r_1 = r_2)$

This paper is outlined as follows. Section 2 provides some related work in detecting model inconsistencies. In section 3, we describe the process through which possible model inconsistencies are detected. Sections 4 and 5 provide procedures for translating rules contained in metamodel specifications into first order predicate logic. Section 6 describes how model elements are included in the transformation. Section 7 provides an example of inconsistency detection using our method. Finally section 8 summarizes our method of inconsistency detection.

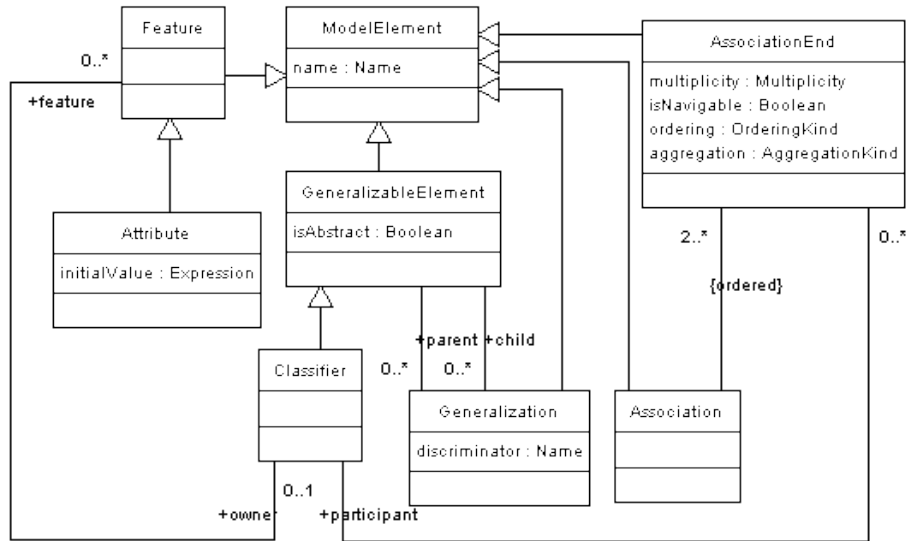


Fig. 1. A simplified subset of the UML class diagram definition

2 Related Work on Inconsistency Detection

Previous work on model consistency management has mainly focused on three different categories of approaches:

- In the first category, the focus has been on enhancing the metamodel to make it as precise as possible. For example in [19] the aim has been to identify areas of ambiguity in UML and defining precise semantics for it. In [20] a sufficiently expressive subset of UML has been selected and formal semantics has been specified for this subset.
- In the second category, the aim has been on increasing the expressiveness of the metamodel constraint definition language (i.e. OCL). For example in [21] and [22] extensions to OCL have been proposed.
- The third and widest category of approaches focuses on translating models into a known formal language. For example in [11] a restricted subset of UML class diagrams and some common inconsistencies have been considered. These predetermined inconsistencies are identified by translating their semantics into first order predicate logic. In [6] consistency checking between UML class diagrams and sequence diagrams is considered. Main relations between these diagrams are expressed as typed and attributed graphs and graph grammars, and algorithms for inconsistency checking based on those graphs have been explained. By now a great number of papers have focused on translating UML models into a language with known semantics and available tools, for example: description logics [7], the Maude language [13], the

Mathematical System Model (MSM) [14], the Z specification language [15] and its extension Object Z [16], EER diagrams [17] and the logical language of PVS [18]. These methods focus on a specific type of UML model (e.g. class diagram) and can only detect foreseen inconsistency types.

3 The Inconsistency Detection Process

As stated earlier, our method for detecting model inconsistencies takes as input the metamodel diagrams along with OCL well-formedness rules. These are translated into first-order predicate logic (FoPL) and provide the initial logic database. With the metamodel logic rules at hand, the next step is to translate the model (to be validated) into FoPL terms. Figure 2 shows how this process is done.

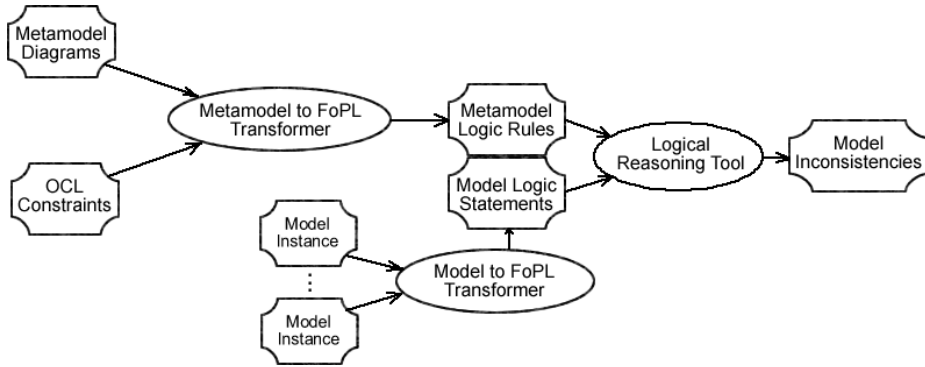


Fig. 2. The Inconsistency Detection Process

Our FoPL translation consists of a set of terms, predicates, quantifiers and formulas. The set of types, predicates, functions and relation symbols are denoted by S . The set of formulas generated from translating OCL well-formedness rules are denoted by F . In the following sections for simplicity of notation, when defining logic functions, we will use ' $f : T_1 \rightarrow T_2$ ' to define function f and its type conformance rule (where T_1 and T_2 are predicate types added to S in translating metamodel types, see section 4):

$$\forall arg, ret (ret = f(arg) \rightarrow T_1(arg) \wedge T_2(ret))$$

In the following two sections, a precise translation from metamodel diagrams and OCL well-formedness rules into FoPL is presented. We will concentrate on those parts of the metamodel which are relevant to inconsistency detection. Any logical reasoning tool can then be used to find possible inconsistencies or to verify that the target model is consistent according to metamodel rules.

4 Translating Metamodel Diagrams

Metamodel diagrams are schemas that define rules and constructs for model creation. Such a schema exists for each type of UML diagram (e.g. class diagrams, state diagrams, ...). These schemas conform to the rules specified in OMG's MOF specification [24] and contain five main elements: Class, Association, Generalization (specified as a special property called superclass in the Class element), Property and Operation. In the following, a first order logic translation for each element is presented:

- For every metamodel class, a predicate with the same name is added to S. For example, looking at figure 1 which represents a subset of the UML class diagram definition, this translation will introduce 'ModelElement', 'Attribute', 'AssociationEnd', ... as new predicate symbols in S. We also include a predicate type in S for each MOF predefined type such as Integer, Boolean, Multiplicity, etc. In order to handle collection types, for each predicate type T in S, predicate types Set_T , $Sequence_T$ and Bag_T will also be included in S.
- For binary association ends from class C_s to class C_d function ' $f : C_s \rightarrow C_d$ ' is added to S (as stated earlier this notation means that we will also add a type conformance rule to F:

$$\forall arg, ret (ret = f(arg) \rightarrow C_s(arg) \wedge C_d(ret))$$

In the following, we will skip adding these type conformance rules and use the shorter notation instead). If the multiplicity at side C_d is more than one, ' $f : C_s \rightarrow Set_{C_d}$ ' is introduced instead. If the C_d side is ordered then ' $f : C_s \rightarrow Sequence_{C_d}$ ' is introduced. If an association end represents an n-ary association, in addition, we have to add an n-ary predicate to S. Looking at figure 1, our translation method will add ' $owner : Feature \rightarrow Classifier$ ' and ' $feature : Classifier \rightarrow Set_{Feature}$ ' along with other similar functions to S (In order to have unique function names, some naming convention must be adopted. For example we can use 'Feature_Classifier_owner' instead of just 'owner' as a function name to provide uniqueness. But for simplicity we will ignore this issue here and use the shorter term, i.e. owner). This translation will also add the following rule to specify the relationship between two sides of the association:

$$\forall c, f (Classifier(c) \wedge Feature(f) \wedge c = owner(f) \leftrightarrow f \in feature(c))'$$

- Generalizations will be handled in the instantiation of types included in S. If C_s is a subclass of C_p then: ' $\forall c (C_s(c) \rightarrow C_p(c))$ '.
- For every property with type T in a metamodel class C this translation adds a function ' $f : C \rightarrow T$ ' to S. For example in figure 1 the isAbstract property of class GeneralizableElement will add ' $isAbstract : GeneralizableElement \rightarrow Boolean$ ' to S. (Again, the class name can be attached to the property name to provide a unique function name). Operations of a metamodel class will have a similar translation, except that parameter types of the operation will be included in the function domain.

5 Translating OCL Constraints

In this section we present a procedure for translating OCL expressions into FoPL. We will focus on OCL features relevant to defining metamodel well-formedness rules and skip non-relevant features. Each expression in OCL is written in the context of an instance of a type (a metamodel class in our case). OCL expressions can be part of an invariant, operation body, precondition, postcondition, initial or derived value. The procedure provided in this section can be used to translate all these types of expressions in a similar way (although the latter four expression types have limited usage in defining metamodel constraints). In the following, we will use $[exp]$ to denote the first order translation of OCL expression ' exp '.

5.1 Translating Invariants

An invariant of a type in OCL must be true for all instances of that type at any time. OCL invariants have the following syntax: 'Context C inv: i '. For each invariant we will add the following formula to F: ' $\forall c (C(c) \rightarrow [i])$ '.

5.2 Translating Basic Values and Types

The basic types of OCL are Boolean, Integer, Real and String [25]. As mentioned earlier, for each OCL predefined type we will add the corresponding logic type to S. OCL defines a number of operations on these types (such as *,+,- for Integer and Real, or and implies for Boolean and size for String). Boolean operators are translated to the corresponding FoPL operators. The other predefined operations will be handled by adding functions to S (for example we will add ' $size : String \rightarrow Integer$ ' to S).

5.3 Predefined OCL Properties on All Objects

There are several properties in OCL that apply to all objects. These include `oclIsTypeOf` and `oclAsType` which are frequently used in defining metamodel constraints. The operation `oclIsTypeOf` deals with the type of an object. Any expression of the form ' $exp.oclIsTypeOf(T)$ ' can be translated into ' $T([exp])$ '.

The operation `oclAsType` takes as argument an object of type T_1 and returns an object of type T_2 which is a subtype of T_1 . The translation of ' $v.oclAsType(T_2)$ ' includes adding a new function ' $oclAsType_{T_1,T_2} : T_1 \rightarrow T_2$ ' to S.

5.4 Translating Objects and Properties

In OCL a property is an attribute, association end or an operation call with `isQuery` being true.

- Attributes and association ends with multiplicity one are accessed by using the '.' operator. Any OCL expression of the form ' $exp.a$ ' can be translated to ' $a([exp])$ ' (remember that we have previously added function 'a' to S in translating metamodel diagrams).

- In OCL, navigation through association ends with multiplicity more than one results in a collection instance. The type *Collection* is predefined in OCL. It contains a large number of predefined operations to manipulate collections. *Collection* is an abstract type with *Set*, *Sequence* and *Bag* as its subtypes. In the next subsection we describe the first order translation of the features common to all collection types.
- In OCL an operation can be defined by a precondition constraint. The operation's return value can be referred to by *result*. It takes the following form:
Context $C :: op(p_1 : T_1, \dots, p_n : T_n) : returnType$
 $post : result = exp$
For every operation definition inside the metamodel specification we will add a function with the same name and signature to *S*. In addition, we will include the following formula:
' $forall\ c, p_1, \dots, p_n\ (C(c) \wedge T_1(p_1) \wedge \dots \wedge T_n(p_n) \rightarrow op(c, p_1, \dots, p_n) = [exp])$ '
Examples of operation definitions in the UML specification include 'opposite-AssociationEnds' and 'allAttributes' defined in *Classifier* and 'upperbound' defined in *AssociationEnd*.
- The syntax of OCL operation calls is similar to that of attributes with the exception that operations can have parameters. Any operation call of the form ' $exp.op(p_1, \dots, p_n)$ ' is translated into the first order formula ' $op([exp], [p_1], \dots, [p_n])$ '.

5.5 Translating Collection Predefined Operations

In the following, we describe the first order translation of the features common to all collection types. we will use symbol $Collection_T$ in our first order translation to denote any of the three collection predicate types: Set_T , $Sequence_T$ and Bag_T (where *T* is the type of elements contained in the collection).

- *Translating select, reject and collect*: OCL has special constructs to specify a selection from a specific collection. These are *select*, *reject* and *collect*. *Select* keeps a subset from a collection for which the input expression evaluates to true. *Reject* does the opposite and keeps the subset for which the input expression evaluates to false. For an OCL expression in the form of ' $c \rightarrow select(v|exp)$ ' we will add the function ' $select_{v|exp} : Collection_T \rightarrow Collection_T$ ' to *S* and add the following formula to *F*:
' $\forall c, v\ (Collection_T(c) \wedge T(v) \wedge [exp] \leftrightarrow v \in select_{v|exp}(c))$ '
A similar translation can be used for *reject*. The *collect* operation is similar to *select* except that it does not extract the items themselves from the source collection, instead it selects properties of collection items. For example, if *c* is a collection of *Feature* instances (see figure 1), ' $c \rightarrow collect(f|f.owner)$ ' will result in a collection of *Classifier* (owners of those *Feature* instances). As with *select* and *reject*, for an OCL expression in the form of ' $c \rightarrow collect(v|v.a)$ ' we will add the function ' $collect_{v|v.a} : Collection_T \rightarrow Collection_{T_2}$ ' to *S* (where *T* is the type of elements of *c* and *T*₂ is the type of *v.a*). In addition, we will include the following formula in *F*:
' $\forall c, v\ (Collection_T(c) \wedge T(v) \leftrightarrow a(v) \in collect_{v|v.a}(c))$ '

- *Translating forAll*: the forAll operation in OCL allows specifying a boolean expression, which must hold for all objects in a collection. Any expression of the form ' $c \rightarrow \text{forAll}(v|\text{exp})$ ' can be translated into ' $\forall v (T(v) \wedge v \in [c] \rightarrow [\text{exp}])$ '.
- *Translating exists*: the exists operation in OCL allows specifying a boolean expression that must hold for at least one object in a collection. Any expression of the form ' $c \rightarrow \text{exists}(v|\text{exp})$ ' can be translated into: ' $\exists v (T(v) \wedge v \in [c] \wedge [\text{exp}])$ '.
- *Translating size, union, sum and count*: for each of these OCL operations, a function with the same name is added to S. For example ' $\text{size} : \text{Collection}_T \rightarrow \text{Integer}$ ' is added to F and ' $c \rightarrow \text{size}$ ' is translated into ' $\text{size}([c])$ '.
- *Translating includes, includesAll, excludes and excludesAll*: These OCL operations test whether a collection contains an element or another collection. Any expression of the form ' $c \rightarrow \text{includes}(v)$ ' is translated into ' $v \in [c]$ '. Any expression of the form ' $c \rightarrow \text{includesAll}(c_2)$ ' is translated into: ' $\forall v (T(v) \wedge v \in [c_2] \rightarrow v \in [c])$ '. Similar translations can be used for excludes and excludesAll.

6 Including Model Elements in the Translation

The goal of our method is to check the consistency of models against rules and constraints provided in the metamodel specification. We have included metamodel rules in our logic database. What remains, is to provide a transformation from the target model to FoPL. Model elements are instances of metamodel diagram classifiers. For example in figure 3 Book, Author and Publisher are instances of Classifier. There are also two instances of Attribute, two instances of Association and four instances of AssociationEnd. To translate this model, we have to add the following declarations to our logic database (as stated earlier we have to provide a naming mechanism to ensure that instance names will be unique. For example we have used 'Book_title' instead of just 'title' in our logic naming to avoid confusion with attributes in other classes):

```

Classifier(Book).
Classifier(Author).
Classifier(Publisher).
Attribute(Book_title).
Attribute(Author_name).
Association(Association1).
Association(Association2).
AssociationEnd(Book_Publisher_publisher).
AssociationEnd(Publisher_Book_pubBooks).
AssociationEnd(Book_Auhtor_author).
AssociationEnd(Author_Book_authBooks).
participant(Book_Publisher_publisher) = Publisher.
participant(Book_Auhtor_author) = Author.
participant(Publisher_Book_pubBooks) = Book.

```

```

participant(Author_Book_authBooks) = Book.
owner(Book_title) = Book.
owner(Author_name) = Author.
name(Book_Publisher_publisher) = 'publisher'.
name(Publisher_Book_pubBooks) = 'pubBooks'.
name(Book_Auhtor_author) = 'author'.
name(Author_Book_authBooks) = 'authBooks'.
name(Book_title) = 'title'.
name(Author_name) = 'name'.

```

In the above declarations, participant, owner and name are functions defined from the class diagram definition of figure 1 according to the translations specified in section 3.

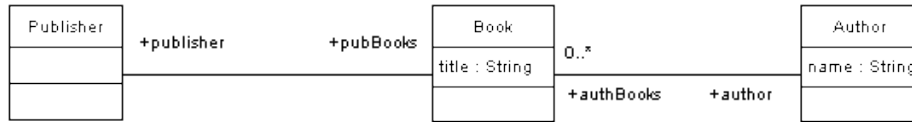


Fig. 3. An instance class diagram

7 An Example of Inconsistency Detection

In this section we provide an example of detecting a common inconsistency (i.e. opposite association ends with the same name for a class) in class diagrams, using the method presented in this paper. Figure 4 shows a modified version of the class diagram instance of figure 3 where Book has two opposite association ends connected to Author and Publisher both named 'stakeholder'. This is an

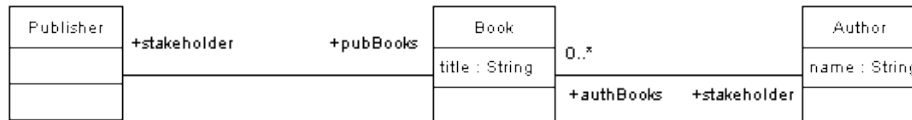


Fig. 4. An inconsistent class diagram instance

inconsistency according to the following OCL constraint:

Context Classifier inv:

$$\begin{aligned} & self.allOppositeAssociationEnds \rightarrow \\ & \quad forAll(p, q | p.name = q.name \text{ implies } p = q) \end{aligned}$$

The above constraint can be translated into the following logic rule which will be added to F (see section 5):

$$\begin{aligned} & \forall p, q, r (AssociationEnd(p) \wedge AssociationEnd(q) \wedge Classifier(r) \\ & \quad \rightarrow (p, q \in allOppositeAssociationEnds(r)) \\ & \quad \rightarrow (name(p) = name(q) \rightarrow p = q \vee inconsistency(p, q)) \end{aligned} \tag{1}$$

We have explicitly added 'inconsistency(p,q)' in the above translation to detect possible inconsistencies. The way this rule is added depends on the kind of logical reasoning tool being used. For example one can use alerting mechanisms specified in a logical reasoning tool (e.g. SWI Prolog) to alert the user when an inconsistency is detected.

The above OCL constraint uses function 'allOppositeAssociationEnds' which is defined in the context of Classifier as:

$$\begin{aligned} & Context Classifier :: allOppositeAssociationEnds() : Set(AssociationEnd) \\ & \quad post : result = self.oppositeAssociationEnds \rightarrow \\ & \quad \quad union(self.parent.allOppositeAssociationEnds) \end{aligned}$$

To complete our translation, we have to add 'allOppositeAssociationEnds : Classifier \rightarrow Set_{AssociationEnd}' as a function to S. We will also add the following rule to F (see section 5.4):

$$\begin{aligned} & \forall c (Classifier(c) \rightarrow allOppositeAssociationEnds(c) = \\ & \quad union(oppositeAssociationEnds(c), allOppositeAssociationEnds(parent(c))) \end{aligned}$$

Note that this definition uses operation 'oppositeAssociationEnds' which is also defined as an OCL operation in UML specifications. In order to complete our translation we must also add the translation for that operation to our logic database. This is done in the same way we have defined allOppositeAssociationEnds. For simplicity we will skip this definition here. To complete this process we also have to include model semantics into our translation. For this purpose, we use the same declarations provided in section 6 (for the class diagram instance of figure 3) with the following changes:

$$\begin{aligned} & AssociationEnd(Book_Publisher_stakeholder). \\ & AssociationEnd(Book_Author_stakeholder). \\ & name(Book_Publisher_publisher) = 'stakeholder'. \\ & name(Book_Author_stakeholder) = 'stakeholder'. \end{aligned}$$

By using a reasoning tool we can easily extract inconsistencies from the above rules:

$$name(Book_Publisher_stakeholder) = stakeholder \tag{2}$$

$$\text{name}(\text{Book_Author_stakeholder}) = \text{stakeholder} \quad (3)$$

$$\text{Book_Publisher_stakeholder} \langle \rangle \text{Book_Author_stakeholder} \quad (4)$$

(1), (2), (3), (4) \implies

inconsistency(Book_Publisher_stakeholder, Book_Author_stakeholder).

8 Conclusions

In this paper, we presented a method for inconsistency detection at the meta-model level. The precise semantics of the metamodel along with a target (possibly inconsistent) model are the inputs to this method. We introduced a translation to first order predicate logic for each part of the input. The main advantage of this method is that it is not restricted to any specific type of metamodel diagram. It can be applied to class diagrams, state diagrams and other UML diagrams (as well as any other metamodel instance) as long as the metamodel specifications for that diagram type are available. This method (unlike previous inconsistency detection methods) is not restricted to predetermined inconsistencies and it can detect any error according to metamodel rules.

References

1. Alvarez, A., Aleman, J.: Formally modeling UML and its evolution: A holistic approach. In S. Smith and C. Talcott, editors, Proceedings, Formal Methods for Open Object based Distributed Systems, Stanford, USA, pages 183-206. Kluwer, 2000
2. Bodeveix, J.P., Millan, T., Percebois, C., Le Camus, C., Bazex, P., Feraud, L., Sobek, R.: Extending OCL for Verifying UML Models Consistency. Workshop on Consistency Problems in UML-based Software Development, 5th International Conference on the Unified Modeling Language- the Language and its applications (UML'2002), Dresden, Germany 75-90, 2002
3. Breu, R., Gosu, R., Huber, F., Rumpe, B., Schwerin, W.: Towards a precise semantics for object-oriented modeling techniques. In J. Bosch and S. Mitchell, editors, ECOOP Workshop, Jyvaskyla, Finland, LNCS 1357, pages 205-210. Springer, 1998
4. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. J. Syst. Software, 2009
5. Cabot, J., Teniente, E.: Transformation techniques for OCL constraints. Science of Computer Programming 68 (3), 2007
6. Cabot, J.: From declarative to imperative UML/OCL operation specifications. In: Proc. 26th Int. Conf. on Conceptual Modeling (ER 2007) LNCS, vol.4801, pp.198213, 2007
7. Ehrig, H., Tsiolakis, A.: Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig & G. Taentzer (Eds.), ETAPS 2000 Workshop on Graph Transformation Systems (pp.77-86), Berlin, Germany, 2000
8. Evans, A., Kent, S.: Meta-modeling semantic of UML: the pUML approach. In proceedings of UML 1999. LNCS 1723, 140-155, 1999

9. France, R.: A problem-oriented analysis of basic UML static modeling concepts. In Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, USA, volume 34 (10) of ACM SIGPLAN notices. ACM Press, 1999
10. Garcia, M.: Efficient integrity checking for essential MOF + OCL in software repositories. *Journal of Object Technology* 7 (6), 101, 2008
11. Garcia, M., Fuentes-Fernandez, R., Gomez-Sanz, J.: Guideline for the definition of EMF metamodels using an Entity-Relationship approach, *Inform. Softw. Technol.*, 2009
12. Gogolla, M., Richters, M.: On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language: Technical Aspects and Applications*, pages 109-121. Physica-Verlag, 1998
13. Hooman, J.: Towards Formal support for UML-based development of embedded systems. University of Nijmegen. Available at: http://www-omega.imag.fr/doc/d1000230_1/HoomanPROGRESS.pdf
14. Kaneiwa, K., Satoh, K.: Consistency Checking Algorithms for Restricted UML Class Diagrams. Proceedings of the Fourth International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2006), LNCS 3861, pp. 219-239, Budapest, Hungary, 2006
15. Kim, S., Carrington, D.: Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *Proceedings, Unified Modeling Language*, Fort Collins, USA, LNCS 1723, pages 83-98. Springer, 1999
16. Krishnan, P.: Consistency checks for UML. In *Proceedings, Asia Pacific Software Engineering Conference (APSEC)*, pages 162-169, 2000
17. Kuhne, T.: Matters of (Meta-) Modeling. *Journal on Software and Systems Modeling*, Volume 5, Number 4, pp. 369-385, 2006
18. Loecher, S., Ocke, S.: A Metamodel-Based OCL-Compiler for UML and MOF, *Electronic Notes in Theoretical Computer Science* 102 (2004) 4361
19. Mens, T., Van Der Straeten, R., Simmonds, J.: A Framework for Managing Consistency of Evolving UML Models. In *Software Evolution with UML and XML*, IDEA Group Publishing, 2005
20. Object Management Group, *Unified Modeling Language (UML), Infrastructure, V2.2*, 2009, <http://www.omg.org>
21. Object Management Group: *Meta Object Facility (MOF) Core Specification, v2.0*, 2006, <http://www.omg.org>
22. Object Management Group: *Object Constraint Language (OCL), v2.0*, 2006, <http://www.omg.org>
23. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, 1, pp. 329-380, World Scientific Publishing Co., 2001
24. Warmer, J., Kleppe, A.: Extending OCL to Include Actions. In *Proceedings of UML 2000*. p. 440-450, Springer-Verlag, 2000