Consistency Checking Algorithms for Restricted UML Class Diagrams

Ken Kaneiwa and Ken Satoh

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
{kaneiwa,ksatoh}@nii.ac.jp

Abstract. Automatic debugging of UML class diagrams helps in the visual specification of software systems because users cannot detect errors in logical inconsistency easily. This paper focuses on *tractable* consistency checking of UML class diagrams. We accurately identify inconsistencies in these diagrams by translating them into first-order predicate logic generalized by counting quantifiers and classify their expressivities by eliminating some components. For class diagrams of different expressive powers, we introduce optimized algorithms that compute their respective consistencies in P, NP, PSPACE, or EXPTIME with respect to the size of a class diagram. In particular, for two cases in which class diagrams contain (i) disjointness constraints and overwriting/multiple inheritances and (ii) these components along with completeness constraints, the restriction of attribute value types decreases the complexities from EXPTIME to P and PSPACE. Additionally, we confirm the existence of a meaningful restriction of class diagrams that prevents any logical inconsistency.

1 Introduction

The Unified Modeling Language (UML) [11, 6] is a standard modeling language; it is used as a visual tool for designing software systems. However, visualized descriptions make it difficult to determine consistency in formal semantics. In order to design UML diagrams, designers check not only for syntax errors but also for *logical inconsistency*, which may be present implicitly in the diagrams. Automatic detection of errors is very helpful for designers; for example, it enables them to revise erroneous parts of UML diagrams by determining inconsistent classes or attributes. Moreover, in order to confirm the accuracy of debugging (soundness, completeness, and termination), a consistency checking algorithm should be developed computationally and theoretically.

Class diagrams, which are a type of UML diagrams, are employed to model concepts in static views. The consistency of class diagrams has been investigated as follows. Evans [5] attempted a rigorous description of UML class diagrams by using the Object Constraint Language (OCL) and treated UML reasoning. Beckert, Keller, and Schmitt [1] defined a translation of UML class diagrams with OCL into first-order predicate logic. Further, Tsiolakis and Ehrig [13] analyzed

the consistency of UML class and sequence diagrams by using attributed graph grammars. The OCL and other approaches provide rigorous semantics and logical reasoning on UML class diagrams; however, they do not theoretically analyze the worst-case complexity of consistency checking. On the other hand, a number of object-oriented models and their consistency [10, 12] have been considered for developing software systems, but the models do not characterize the components of UML class diagrams; for example, the semantics of attribute multiplicities is not supported.

Berardi, Calvanese, and De Giacomo presented the correspondence between UML class diagrams and description logics (DLs), which enables us to utilize DL-based systems for reasoning on UML class diagrams [2]. In fact, Franconi and Ng implemented the concept modeling system ICOM [7] using DLs. The cyclic expressions of class diagrams are represented by *general axioms* for DLs. For example, a class diagram is cyclic if a class C has an attribute and the type of the attribute value is defined by the same class. However, it is well known that reasoning on general axioms of the necessary DLs is exponential time hard [3]. Therefore, consistency checking of the class diagrams in DLs requires exponential time in the worst case.

In order to reduce the complexity, we consider restricted UML class diagrams obtained by deleting some components. A meaningful restriction of class diagrams is expected to avoid intractable reasoning, thus facilitating automatic debugging. This solution provides us with not only tractable consistency checking but also a sound family of class diagrams (i.e., its consistency is theoretically guaranteed without checking).

The aim of this paper is to present optimized algorithms for testing the consistency of restricted UML class diagrams, which are designed to be suitable for class diagrams of different expressive powers. The algorithms detect the logical inconsistency of class diagram formulation in first-order predicate logic generalized by counting quantifiers [9]. Although past approaches employ reasoning algorithms of DL and OCL, we develop consistency checking algorithms specifically for UML class diagrams. Our algorithms deal directly with the structure of UML class diagrams; hence, they enable the following:

- Easy recognition of the inconsistency triggers in the diagram structure, such as combinations of disjointness/completeness constraints, attribute multiplicities, and overwriting/multiple inheritances, and
- Refinement of the algorithms when the expressivity is changed by the presence of the inconsistency triggers.

The inconsistency triggers captured by the diagram structure are used to restrict some relevant class diagram components in order to derive a classification of UML class diagrams. Since we can theoretically prove that there arises no inconsistency of eliminated components, the algorithms will become simplified and optimized for their respective expressivity.

The contributions of this paper are as follows:

1. *Inconsistency triggers:* We accurately identify the inconsistency triggers that cause logical inconsistency among classes, attributes, and associations.

2. *Expressivity:* We classify the expressivity of UML class diagrams by deleting and adding certain inconsistency triggers.
3. *Algorithms and complexities:* We develop several consistency checking algorithms for class diagrams of different expressive powers and demonstrate that they compute the consistency of those class diagrams in P, NP, PSPACE, or EXPTIME with respect to the size of a class diagram.
4. *Tractable consistency checking in the optimized algorithms:* When the attribute value types are defined with restrictions in class diagrams, consistency checking is respectively computable in P and PSPACE for two cases in which the diagrams contain (i) disjointness constraints and overwriting/multiple inheritances and (ii) these components with completeness constraints.
5. *Consistent class diagrams:* We demonstrate that every class diagram is consistent if the expressivity is restricted by deleting disjointness constraints and overwriting/multiple inheritances (but allowing attributes multiplicities and simple inheritances). Thus, we need not test the consistency of such less expressive class diagrams ($\mathcal{D}_0^-$ and $\mathcal{D}_{com}^-$).

There are two main advantages with regard to the results of this study. First, the optimized algorithms support efficient reasoning for various expressive powers of class diagrams. In contrast, the DL formalisms do not provide optimized algorithms for the *restricted* UML class diagrams because general axioms of DLs require exponential time even if DLs are restricted [3]. Therefore, the classification of DLs does not fit into the classification of UML class diagrams[1]. Second, a meaningful restriction of UML class diagrams is analyzed. We confirm the existence of restricted class diagrams that permit attribute multiplicities but that cause no logical inconsistency.

2 Class Diagrams in FOPL with Counting Quantifiers

We define a translation of UML class diagrams into first-order predicate logic generalized by counting quantifiers. The reasons for encoding into first-order predicate logic with counting quantifiers are as follows. First, the semantics of UML class diagrams should be defined by encoding them in a logical language because consistency checking is based on the semantics of encoded formulas. In other words, no consistency checking algorithm can operate on original diagrams without formal semantics. Second, variables and quantifiers in first-order logic lead to an explicit formulation that is useful to restrict/classify the expressive powers. In contrast, DL encoding [2] conceals the quantification of variables in expressions.

The alphabet of UML class diagrams consists of a set of class names, a set of attribute names, a set of operation names, a set of association names, and a set of datatype names. Let C, C', C_i be class names, a, a' attribute names, f, f'

[1] Note that reasoning on general axioms becomes exponential hard even if the small DL $\mathcal{AL}$ contains no disjunction, qualified existential restriction, and number restriction.

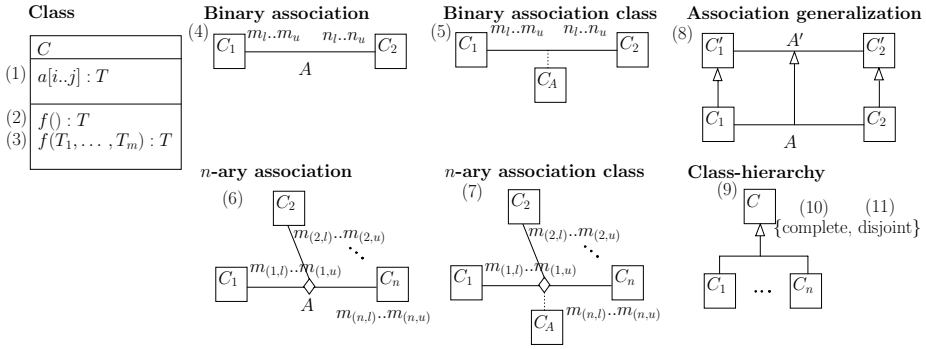


Fig. 1. Components of UML class diagrams

operation names, A, A' association names, and t, t', t_i datatype names. Let type T be either a class or a datatype. The leftmost figure in Fig.1 represents a class C with an attribute $a[i..j]\colon T$, a 0-ary operation $f()\colon T$, and an n-ary operation $f(T_1, \ldots, T_n)\colon T$, where $[i..j]$ is the attribute multiplicity and T and $T_1, \ldots, T_n$ are types. Any class C can be expressed as the unary predicate C in first-order logic. Let F_1 and F_2 be first-order formulas. We denote the implication form $F_1 \to F_2$ as the universal closure $\forall x_1 \cdots \forall x_n.(F_1 \to F_2)$ where $x_1, \ldots, x_n$ are all the free variables occurring in $F_1 \to F_2$. Let $F(x)$ denote a formula F in which the free variable x occurs. The counting quantifier formula $\exists_{\geq i} x.F(x)$ implies that at least i elements x satisfy $F(x)$, while the counting quantifier formula $\exists_{\leq i} x.F(x)$ implies that at most i elements x satisfy $F(x)$. The value type T and multiplicity $[i..j]$ of the attribute a in the class C are specified by the following implication forms:

(1) $C(x) \to (a(x, y) \to T(y))$ and $C(x) \to \exists_{\geq i} z.a(x, z) \wedge \exists_{\leq j} z.a(x, z)$

where a is a binary predicate and T is a unary predicate. Moreover, the 0-ary operation $f()\colon T$ of the class C is specified by the following implication forms:

(2) $C(x) \to (f(x, y) \to T(y))$ and $C(x) \to \exists_{\leq 1} z.f(x, z)$

where f is a binary predicate and T is a unary predicate. The n-ary operation $f(T_1, \ldots, T_n)\colon T$ of the class C is specified by the following implication forms:

(3) $C(x) \to (f(x, y_1, \ldots, y_n, z) \to T_1(y_1) \wedge \cdots \wedge T_n(y_n) \wedge T(z))$
$C(x) \to \exists_{\leq 1} z.f(x, y_1, \ldots, y_n, z)$

where f is an $n+2$-ary predicate and each T_i, T are unary predicates.

We next formalize associations A that imply connections among classes $C_1, \ldots, C_n$ (as in (4) and (6) of Fig.1). A binary association A between two classes C_1 and C_2 and the multiplicities $m_l..m_u$ and $n_l..n_u$ are specified by the forms:

(4) $A(x_1, x_2) \to C_1(x_1) \wedge C_2(x_2)$

$C_1(x) \to \exists_{\geq n_l} x_2.A(x, x_2) \wedge \exists_{\leq n_u} x_2.A(x, x_2)$
$C_2(x) \to \exists_{\geq m_l} x_1.A(x_1, x) \wedge \exists_{\leq m_u} x_1.A(x_1, x)$

where A is a binary predicate and C_1, C_2 are unary predicates. In addition to the formulas, if an association is represented by a class, then the association class C_A is specified by supplementing the implication forms below:

(5) $A(x_1, x_2) \to (r_0(x_1, x_2, z) \to C_A(z))$
$A(x_1, x_2) \to \exists_{=1} z.r_0(x_1, x_2, z)$ and $\exists_{\leq 1} z.(r_0(x_1, x_2, z) \wedge C_A(z))$

where C_A is a unary predicate and r_0 is a ternary predicate. By extending the formulation of a binary association, the n-ary association A among classes C_1, ... , C_n and their multiplicities "$m_{(1,l)}..m_{(1,u)}$", ... , "$m_{(n,l)}..m_{(n,u)}$" (as shown in (6) of Fig.1) are specified by the following implication forms:

(6) $A(x_1, \ldots, x_n) \to C_1(x_1) \wedge \cdots \wedge C_n(x_n)$
$C_k(x) \to \exists_{\geq m_{(1,l)}} x_1 \cdots \exists_{\geq m_{(k-1,l)}} x_{k-1} \exists_{\geq m_{(k+1,l)}} x_{k+1} \cdots \exists_{\geq m_{(n,l)}} x_n.A(x_1, \ldots, x_n)[x_k/x]$
$C_k(x) \to \exists_{\leq m_{(1,u)}} x_1 \cdots \exists_{\leq m_{(k-1,u)}} x_{k-1} \exists_{\leq m_{(k+1,u)}} x_{k+1} \cdots \exists_{\leq m_{(n,u)}} x_n.A(x_1, \ldots, x_n)[x_k/x]$

where A is an n-ary predicate and $[x_k/x]$ is a substitution of x_k with x. In addition, the association class C_A is specified by adding the implication forms below:

(7) $A(x_1, \ldots, x_n) \to (r_0(x_1, \ldots, x_n, z) \to C_A(z))$
$A(x_1, \ldots, x_n) \to \exists_{=1} z.r_0(x_1, \ldots, x_n, z)$ and $\exists_{\leq 1} z.(r_0(x_1, \ldots, x_n, z) \wedge C_A(z))$

where C_A is a unary predicate and r_0 is an $n+1$-ary predicate. Furthermore, we treat association generalization (not discussed in [2]) such that the binary association A' between classes C_1' and C_2' generalizes the binary association A between classes C_1 and C_2 (as in (8) of Fig.1). More universally, the generalization between n-ary associations A and A' is specified by the following implication forms:

(8)' $A(x_1, \ldots, x_n) \to A'(x_1, \ldots, x_n)$ and $C_1(x) \to C_1'(x)$, ... , $C_n(x) \to C_n'(x)$

where A, A' are n-ary predicates and each C_i, C_j' are unary predicates.

We consider class hierarchies and disjointness/completeness constraints of the classes in hierarchies, as shown in (9), (10), and (11) of Fig.1. A class hierarchy (a class C generalizes classes $C_1, \ldots, C_n$) is specified by the implication forms below:

(9) $C_1(x) \to C(x)$, ... , $C_n(x) \to C(x)$

where C and $C_1, \ldots, C_n$ are unary predicates. The completeness constraint between class C and classes $C_1, \ldots, C_n$ and the disjointness constraint among classes $C_1, \ldots, C_n$ are respectively specified by the implication forms:

(10) $C(x) \to C_1(x) \vee \cdots \vee C_n(x)$

(11) $C_i(x) \to \neg C_{i+1}(x) \wedge \cdots \wedge \neg C_n(x)$ for all $i \in \{1, \ldots, n-1\}$

where C and $C_1, \ldots, C_n$ are unary predicates.

Let D be a UML class diagram. $\mathcal{G}(D)$ is called the translation of D and denotes the set of implication forms obtained by the encoding of D in first-order predicate logic with counting quantifiers (using (1)–(11)).

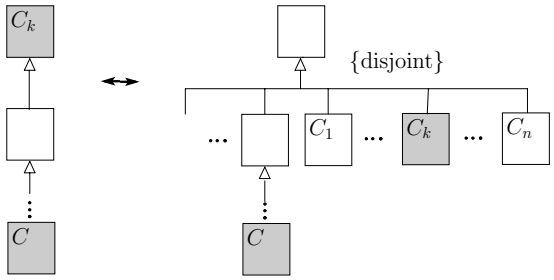
3 Inconsistencies in Class Diagrams

In this section, we analyze inconsistencies among classes, attributes, and associations in UML class diagrams. We first define the syntax errors of duplicate names and irrelevant attribute value types as follows.

Duplicate name errors/attribute value type errors. A UML class diagram D contains a duplicate name error if (i) two classes C_1 and C_2 appear and C_1 and C_2 have the same class name, (ii) two associations A_1 and A_2 appear and A_1 and A_2 have the same association name, or (iii) two attributes a_1 and a_2 appear in a class C and a_1 and a_2 have the same attribute name. Moreover, if two classes have the same name's attributes $a\colon T_1$ and $a\colon T_2$, such that T_1 is a class and T_2 is a datatype, then the class diagram contains an attribute value type error. Obviously, the checking of these syntax errors in a UML class diagram can be computed in linear time.

We elaborate three inconsistency triggers for the UML class diagrams. The reflexive and transitive closure of $\rightarrow$ over classes and associations are denoted by $\rightarrow^*$ such that (i) $C(x) \rightarrow^* C(x)$, (ii) $A(x_1, \ldots, x_n) \rightarrow^* A(x_1, \ldots, x_n)$, (iii) if $C(x) \rightarrow F(x)$, or $C(x) \rightarrow^* C'(x)$ and $C'(x) \rightarrow^* F(x)$, then $C(x) \rightarrow^* F(x)$, and (iv) if $A(x_1, \ldots, x_n) \rightarrow F(x_1, \ldots, x_n)$, or $A(x_1, \ldots, x_n) \rightarrow^* A'(x_1, \ldots, x_n)$ and $A'(x_1, \ldots, x_n) \rightarrow^* F(x_1, \ldots, x_n)$, then $A(x_1, \ldots, x_n) \rightarrow^* F(x_1, \ldots, x_n)$, where $F(x)$ and $F(x_1, \ldots, x_n)$ are any formulas including the free variables.

Inconsistency trigger 1 (generalization and disjointness) The first inconsistency trigger is caused by a combination of generalization and a disjointness constraint. A class diagram has an inconsistency trigger if it contains the formulas $C(x) \rightarrow^* C_k(x)$ and $C(x) \rightarrow^* \neg C_1(x) \wedge \cdots \wedge \neg C_n(x)$ where $1 \leq k \leq n$.



As shown in the above figure, this inconsistency appears when a class C has a superclass C_k but the classes C and C_k are defined as disjoint to each other in the constraint of a class hierarchy.

Inconsistency trigger 2 (overwriting/multiple inheritance) The second inconsistency trigger is caused by one of the following situations:

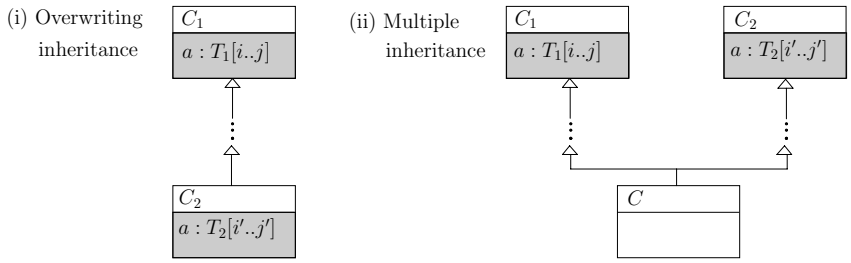
1. (a) conflict between value types T_1 and T_2 when they appear in attributes $a\colon T_1$ and $a\colon T_2$ of the same name, or (b) conflict between multiplicities $[i..j]$ and $[i'..j']$ when they appear in multiplicities $a\colon T_1$ and $a\colon T_2$ of attributes with the same names.

2. conflict between multiplicities when they appear in association and super-associations.

More formally, a class diagram has an inconsistency trigger if it contains a group of the following formulas:

1. $C_2(x) \rightarrow^* C_1(x)$, or $C(x) \rightarrow^* C_1(x)$ and $C(x) \rightarrow^* C_2(x)$, together with
   (a) **Attribute value types:** $C_1(x) \rightarrow (a(x,y) \rightarrow T_1(y))$ and $C_2(x) \rightarrow (a(x,y) \rightarrow T_2(y))$ where T_1 and T_2 are disjoint[2], or
   (b) **Attribute multiplicities:** $C_1(x) \rightarrow \exists_{\geq i} z.a(x,z) \wedge \exists_{\leq j} z.a(x,z)$ and $C_2(x) \rightarrow \exists_{\geq i'} z.a(x,z) \wedge \exists_{\leq j'} z.a(x,z)$ where $i > j'$.

2. **Association multiplicities:** $A(x_1, \ldots, x_n) \rightarrow A'(x_1, \ldots, x_n)$ with $C_k(x) \rightarrow \exists_{\geq m_{(1,l)}} x_1 \cdots \exists_{\geq m_{(k-1,l)}} x_{k-1} \exists_{\geq m_{(k+1,l)}} x_{k+1} \cdots \exists_{\geq m_{(n,l)}} x_n.A(x_1, \ldots, x_n)[x_k/x]$ and $C'_k(x') \rightarrow \exists_{\leq m'_{(1,u)}} x'_1 \cdots \exists_{\leq m'_{(k-1,u)}} x'_{k-1} \exists_{\leq m'_{(k+1,u)}} x'_{k+1} \cdots \exists_{\leq m'_{(n,u)}} x'_n.\ A'(x'_1, \ldots, x'_n)[x'_k/x']$ where $m_{(i,l)} > m'_{(i,u)}$.

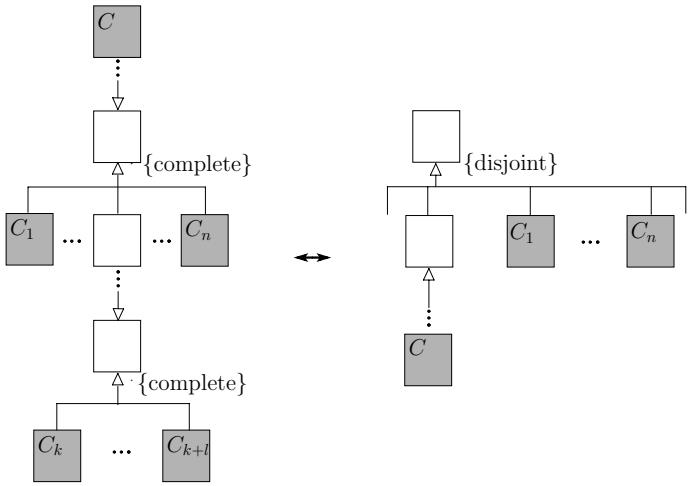


This figure explains that (i) a class C_2 with an attribute $a\colon T_2[i'..j']$ inherits the same name's attribute $a\colon T_1[i..j]$ from a superclass C_1 and (ii) a class C inherits the two attributes $a\colon T_1[i..j]$ and $a\colon T_2[i'..j']$ of the same name from superclasses C_1 and C_2. The former is called *overwriting inheritance*; the latter, *multiple inheritance*. In these cases, if the attribute value types T_1 and T_2 are disjoint or if the multiplicities $[i..j]$ and $[i'..j']$ conflict with each other, then the attributes are determined to be inconsistent. For example, the multiplicities $[1..5]$ and $[10..*]$ cannot simultaneously hold for the same name's attributes.

Inconsistency trigger 3 (completeness and disjointness) A disjointness constraint combined with a completeness constraint can yield the third inconsistency trigger. A class diagram has an inconsistency trigger if it contains the formulas $C(x) \rightarrow^* C_1(x) \vee \cdots \vee C_n(x)$ and $C(x) \rightarrow^* \neg C'_1(x) \wedge \cdots \wedge \neg C'_m(x)$, where $\{C_1, \ldots, C_n\} \subseteq \{C'_1, \ldots, C'_m\}$. This inconsistency appears when classes C and $C_1, \ldots, C_n$ satisfy the completeness constraint in a class hierarchy and classes C and $C'_1, \ldots, C'_m$ satisfy the disjointness constraint in another class hierarchy. Intuitively, any instance of class C must be an instance of one of the classes $C_1, \ldots, C_n$, but each instance of class C cannot be an instance of classes $C'_1, \ldots, C'_m$. Hence, this situation is contradictory.

[2] Types T_1 and T_2 are disjoint if they are classes C_1 and C_2 such that $C_1(x) \rightarrow^* \neg C_2 \in \mathcal{G}(D)$ or if they are datatypes t_1 and t_2 such that $t_1 \cap t_2 = \emptyset$.

The third inconsistency trigger may be more complicated when the number of completeness and disjointness constraints that occur in a class diagram is increased. In other words, disjunctive expressions raised by many completeness constraints expand the search space of finding inconsistency. Let us define the relation $C(x) \rightarrow^+ C_1(x) \vee \cdots \vee C_n(x)$ as follows: (i) if $C(x) \rightarrow^* C_1(x) \vee \cdots \vee C_n(x)$, then $C(x) \rightarrow^+ C_1(x) \vee \cdots \vee C_n(x)$, and (ii) if $C(x) \rightarrow^+ C_1(x) \vee \cdots \vee C_n(x)$ and $C_1(x) \rightarrow^+ DC_1(x)$, $\ldots$, $C_n(x) \rightarrow^+ DC_n(x)$, then $C(x) \rightarrow^+ DC_1(x) \vee \cdots \vee DC_n(x)$ where each DC_i denotes $C'_1(x) \vee \cdots \vee C'_m(x)$ as disjunctive classes. A class diagram has an inconsistency trigger if it contains the formulas $C(x) \rightarrow^+ C_1(x) \vee \cdots \vee C_n(x)$ and for each $i \in \{1, \ldots, n\}$, $C(x) \rightarrow^* \neg C_{(i,1)}(x) \wedge \cdots \wedge \neg C_{(i,m_i)}(x)$, where C_i is one of the classes $C_{(i,1)}, \ldots, C_{(i,m_i)}$. For example, the following figure illustrates that two completeness constraints are complicatedly inconsistent with respect to a disjointness constraint.



The three inconsistency triggers describe all the logical inconsistencies in UML class diagrams if they contain association generalization but not roles. In the next section, we will design a *complete* consistency checking algorithm for finding those inconsistency triggers.

We define a formal model of UML class diagrams using the semantics of FOPL with counting quantifiers. An interpretation $\mathcal{I}$ is an ordered pair (U, I) of the universe U and an interpretation function I for a first-order language.

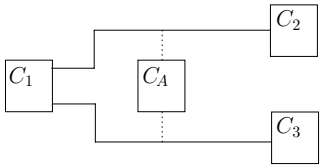
Definition 1 (UML Class Diagram Models). *Let $\mathcal{I} = (U, I)$ be an interpretation. The interpretation $\mathcal{I}$ is a model of a UML class diagram D (called a UML-model of D) if*

1. *$I(C) \neq \emptyset$ for every class C in D and*
2. *$\mathcal{I}$ satisfies $\mathcal{G}(D)$ where $\mathcal{G}(D)$ is the translation of D.*

The first condition indicates that every class is a non-empty class (i.e., an instance of the class exists) and the second condition implies that $\mathcal{I}$ is a first-order

model of the class diagram formulation $\mathcal{G}(D)$. A UML class diagram D is consistent if it has a UML-model.

Furthermore, the following class diagram is invalid because the association class C_A cannot be used for two different binary associations between classes C_1 and C_2 and between classes C_1 and C_3.



Instead of C_A, we describe a ternary association or two association classes. It appears that the EXPTIME-hardness in [2] relies on such expressions. This is because when we reduce (EXPTIME-hard) concept satisfiability in $\mathcal{ALC}$ KBs to class consistency in a UML class diagram, the $\mathcal{ALC}$ KB $\{C_1 \sqsubseteq \exists P_A.C_2,\ C_1 \sqsubseteq \exists P_A.C_3\}$ is encoded into an invalid association class. This condition is important in order to avoid the EXPTIME-hardness and therefore to derive the complexity results in Section 5. This implies that the consistency checking of some restricted UML class diagram groups is computable in P and PSPACE.

4 Consistency Checking

This section presents a consistency checking algorithm for a set of implication forms Γ_0 (corresponding to the UML class diagram formulation $\mathcal{G}(D)$). It consists of two sub-algorithms $Cons$ and $Assoc$: $Cons$ checks the consistency of a class in Γ_0 and $Assoc$ tests the consistency of association generalization in Γ_0.

4.1 Algorithm for Testing Consistency

We decompose an implication form set Γ_0 in order to apply our consistency checking algorithm to it. Let Γ_0 be a set of implication forms, C be a class, and $F_i(x)$ be any formula including a free variable x. Γ is a decomposed set of Γ_0 if the following conditions hold: (i) $\Gamma_0 \subseteq \Gamma$, (ii) if $C(x) \rightarrow F_1(x) \wedge \cdots \wedge F_n(x) \in \Gamma$, then $C(x) \rightarrow F_1(x) \in \Gamma, \ldots, C(x) \rightarrow F_n(x) \in \Gamma$, and (iii) if $C(x) \rightarrow F_1(x) \vee \cdots \vee F_n(x) \in \Gamma$, then $C(x) \rightarrow F_i(x) \in \Gamma$ for some $i \in \{1, \ldots, n\}$. We denote $\Sigma(\Gamma_0)$ as the family of decomposed sets of Γ_0.

We denote $cls(\Gamma_0)$ as the set of classes, $att(\Gamma_0)$ as the set of attributes, and $asc(\Gamma_0)$ as the set of associations that occur in the implication form set Γ_0.

Definition 2. *The following operations will be embedded as subroutines in the consistency checking algorithm:*

1. $H(C, \Gamma) = \{C' \mid C(x) \rightarrow^* C'(x) \in \Gamma\} \cup \{\neg C' \mid C(x) \rightarrow^* \neg C'(x) \in \Gamma\}$.
2. $E(\delta, a, \Gamma) = \bigcup_{C \in \delta} E(C, a, \Gamma)$ *where* $E(C, a, \Gamma) = \{C' \mid C(x) \rightarrow^* (a(x, y) \rightarrow C'(y)) \in \Gamma$ *and* $C(x) \rightarrow^* \exists_{\geq i} z.a(x, z) \in \Gamma\}$ *with* $i \geq 1$.

3. $N(\delta, a, \Gamma) = \bigcup_{C \in \delta} N(C, a, \Gamma)$ *where* $N(C, a, \Gamma) = \{\geq i \mid C(x) \rightarrow^* \exists_{\geq i} z.a(x, z) \in \Gamma\} \cup \{\leq j \mid C(x) \rightarrow^* \exists_{\leq j} z.a(x, z) \in \Gamma\}$.
4. $\mu_0(\delta, \Gamma) = \{C\}$ *if for all* $C' \in \mu(\delta, \Gamma)$, $C \preceq C'$ *and* $C \in \mu(\delta, \Gamma)$ *where* $\mu(\delta, \Gamma) = \{C \in \delta \mid \delta \subseteq H(C, \Gamma)\}$ *and* $\preceq$ *is a linear order over* $cls(\Gamma_0)$.

The operation $H(C, \Gamma)$ denotes the set of superclasses C' of C and disjoint classes $\neg C'$ of C in Γ. The operation $E(\delta, a, \Gamma)$ gathers the set of value types T of attribute a in Γ such that each value type T is of classes in δ. Further, the operation $N(\delta, a, \Gamma)$ gathers the set of multiplicities $\geq i$ and $\leq j$ of attribute a in Γ such that each of these multiplicities is of classes in δ. The operation $\mu(\delta, \Gamma)$ returns a set $\{C_1, \ldots, C_n\}$ of classes in δ such that the superclasses of each C_i (in Γ) subsume all the classes in δ. The operation $\mu_0(\delta, \Gamma)$ returns the singleton set $\{C\}$ of a class in $\mu(\delta, \Gamma)$ such that C is the least class in $\mu(\delta, \Gamma)$ over $\preceq$. The consistency checking algorithm *Cons* is described as follows.

Algorithm *Cons*
input set of classes δ, family of sets of classes Δ, set of implication forms Γ_0
output 1 (consistent) or 0 (inconsistent)
begin
  for $\Gamma \in \Sigma(\Gamma_0)$ **do**
    $S = \bigcup_{C \in \delta} H(C, \Gamma)$; $f_\Gamma = 0$;
    if $\{C, \neg C\} \not\subseteq S$ and $\{t_1, \ldots, t_n\} \not\subseteq S$ s.t. $t_1 \cap \cdots \cap t_n = \emptyset$ **then** $f_\Gamma = 1$;
      for $a \in att(\Gamma_0)$ **do**
        if $i > j$ s.t. $\{\geq i, \leq j\} \subseteq N(\delta, a, \Gamma)$ **then** $f_\Gamma = 0$;
        else $\delta_a = E(\delta, a, \Gamma)$;
          if $\delta_a \neq \emptyset$ and $\delta_a, \mu_0(\delta_a, \Gamma) \not\subseteq \Delta$ **then** $f_\Gamma = Cons(\delta_a, \Delta \cup \{\delta\}, \Gamma_0)$;
        esle
      rof
    fi
    if $f_\Gamma = 1$ **then return** 1;
  rof
  return 0;
end;

In order to decide the consistency of the input implication form set Γ_0, we execute the algorithm $Cons(\{C\}, \emptyset, \Gamma_0)$ for every class $C \in cls(\Gamma_0)$. If C is consistent in Γ_0, it returns 1, else 0 is returned. At the first step of the algorithm, a decomposed set Γ of Γ_0 (in $\Sigma(\Gamma_0)$) is selected, which is one of all the disjunctive branches with respect to the completeness constraints in Γ_0. Subsequently, for each $\Gamma \in \Sigma(\Gamma_0)$, the following three phases are performed.

(1) For the selected Γ, the algorithm checks whether all the superclasses of classes in $\delta = \{C\}$ (obtained from $S = \bigcup_{C \in \delta} H(C, \Gamma)$) are disjoint to each other. Intuitively, it sets a dummy instance of class C and then, the dummy instance is regarded as an instance of the superclasses C' of C and of the disjoint classes $\neg C'$ of C along the implication forms $C(x) \rightarrow^* C'(x)$ and $C(x) \rightarrow^* \neg C'(x)$ in Γ. If an inconsistent pair C_i and $\neg C_i$ possesses the dummy instance, then δ is determined to be inconsistent in Γ. For example, $\{C\}$ is inconsistent in $\Gamma_1 = \{C(x) \rightarrow C_1(x),$

$C_1(x) \rightarrow C_2(x)$, $C(x) \rightarrow \neg C_2(x)\}$ since the inconsistent pair C_2 and $\neg C_2$ must have the dummy instance of the class C, i.e., $H(C, \Gamma_1) = \{C, C_1, C_2, \neg C_2\}$.

(2) If phase (1) finds no inconsistency in Γ, the algorithm next checks the multiplicities of all the attributes $a \in att(\Gamma_0)$. The multiplicities of the same attribute name a are obtained by $N(\delta, a, \Gamma)$; therefore, when $N(\delta, a, \Gamma)$ contains $\{\geq i, \leq j\}$ with $i > j$, these multiplicities are inconsistent. Intuitively, similar to phase (1), the algorithm checks whether superclasses involve conflicting multiplicities along the implication form $C(x) \rightarrow^* C'(x)$ in Γ. For example, $\{C\}$ is inconsistent in $\Gamma_2 = \{C(x) \rightarrow \exists_{\geq 10} z.\ a(x, z),\ C(x) \rightarrow C_1(x),\ C_1(x) \rightarrow \exists_{\leq 5} z.a(x, z)\}$ since the counting quantifiers $\exists_{\geq 10}$ and $\exists_{\leq 5}$ cannot simultaneously hold when $N(\{C\}, a, \Gamma_2) = \{\geq 10, \leq 5\}$.

(3) Next, the disjointness of attribute value types is checked. Along the implication form $C(x) \rightarrow^* C'(x)$ in Γ, the algorithm gathers all the value types of the same name's attributes, obtained by $\delta_a = E(\delta, a, \Gamma)$ for each $a \in att(\Gamma_0)$. For example, $\Gamma_3 = \{C(x) \rightarrow C_1(x),\ C(x) \rightarrow C_2(x),\ C_1(x) \rightarrow (a(x, y) \rightarrow C_3(y)),\ C_2(x) \rightarrow (a(x, y) \rightarrow C_4(y))\}$ derives $\delta_a = \{C_3,\ C_4\}$ by $E(\{C\}, a, \Gamma_3)$ since superclasses C_1 and C_2 of C have the attributes $a\colon C_3$ and $a\colon C_4$. In other words, each value of attribute a is typed by C_3 and C_4. Hence, the algorithm needs to check the consistency of $\delta_a = \{C_3, C_4\}$. In order to accomplish this, it recursively calls $Cons(\delta_a, \Delta \cup \{\{C\}\}, \Gamma_0)$, where δ_a is consistent if 1 is returned. The second argument $\Delta \cup \{\{C\}\}$ prevents infinite looping by storing sets of classes where each set is already checked in the caller processes.

In order to find a consistent decomposed set Γ in the disjunctive branches of $\Sigma(\Gamma_0)$, if the three phases (1), (2), and (3) do not detect any inconsistency in Γ, then the algorithm sets the flag $f_\Gamma = 1$, else it sets $f_\Gamma = 0$. Thus, the flag $f_\Gamma = 1$ indicates that $\{C\}$ is consistent in the input Γ_0, i.e., $Cons(\{C\}, \emptyset, \Gamma_0) = 1$.

In addition to the algorithm $Cons$, the consistency checking of multiplicities over association generalization is processed by the following algorithm $Assoc$. If Γ_0 does not cause any inconsistency with respect to associations, $Assoc(\Gamma_0)$ returns 1, which is computable in polynomial time.

Algorithm *Assoc*
input set of implication forms Γ_0
output 1 (consistent) or 0 (inconsistent)
begin
  for $A \in asc(\Gamma_0)$ and $k \in \{1, \ldots, n\}$ s.t. $arity(A) = n$ **do**
    if $i_v > j_v$ s.t. $\{(\geq i_1, \ldots, \geq i_{k-1}, \geq i_{k+1}, \ldots, \geq i_n),$
      $(\leq j_1, \ldots, \leq j_{k-1}, \leq j_{k+1}, \ldots, \leq j_n)\} \subseteq N_k(H(A, \Gamma_0), \Gamma_0)$ **then return** 0;
  rof
  return 1;
end;

As defined below, the operations $H(A, \Gamma_0)$ and $N_k(\alpha, \Gamma_0)$ respectively return the set of super-associations A' of A and the set of $n-1$-tuples of multiplicities of n-ary associations A in α along the implication forms $C_k(x) \rightarrow \exists_{\geq i_1} x_1 \cdots \exists_{\geq i_{k-1}} x_{k-1}\ \exists_{\geq i_{k+1}} x_{k+1} \cdots \exists_{\geq i_n} x_n.A(x_1, \ldots, x_n)[x_k/x]$ and $C_k(x) \rightarrow \exists_{\leq j_1} x_1 \cdots \exists_{\leq j_{k-1}} x_{k-1}\ \exists_{\leq j_{k+1}} x_{k+1} \cdots \exists_{\leq j_n} x_n.A(x_1, \ldots, x_n)[x_k/x]$, respectively.

Definition 3. *The operations $H(A, \Gamma_0)$ and $N_k(\alpha, \Gamma_0)$ are defined as follows:*

1. $H(A, \Gamma_0) = \{A' \mid A(x_1, \ldots, x_n) \rightarrow^* A'(x_1, \ldots, x_n) \in \Gamma_0\}$.
2. $N_k(\alpha, \Gamma_0) = \bigcup_{A \in \alpha} N_k(A, \Gamma_0)$ *where* $N_k(A, \Gamma_0) = \{(\geq i_1, \ldots, \geq i_{k-1}, \geq i_{k+1}, \ldots, \geq i_n) \mid C_k(x) \rightarrow \exists_{\geq i_1} x_1 \cdots \exists_{\geq i_{k-1}} x_{k-1} \exists_{\geq i_{k+1}} x_{k+1} \cdots \exists_{\geq i_n} x_n . A(x_1, \ldots, x_n)[x_k/x] \in \Gamma_0\} \cup \{(\leq j_1, \ldots, \leq j_{k-1}, \leq j_{k+1}, \ldots, \leq j_n) \mid C_k(x) \rightarrow \exists_{\leq j_1} x_1 \cdots \exists_{\leq j_{k-1}} x_{k-1} \exists_{\leq j_{k+1}} x_{k+1} \cdots \exists_{\leq j_n} x_n . A(x_1, \ldots, x_n)[x_k/x] \in \Gamma_0\}$.

4.2 Soundness, Completeness, and Termination

We sketch a proof of the completeness for the algorithms *Cons* and *Assoc*. Assume that $Cons(\{C\}, \emptyset, \mathcal{G}(D))$ for all $C \in cls(\mathcal{G}(D))$ and $Assoc(\mathcal{G}(D))$ are called. We construct an implication tree of $(C, \mathcal{G}(D))$ that expresses the consistency checking proof of C in $\mathcal{G}(D)$. If $Cons(\{C\}, \emptyset, \mathcal{G}(D)) = 1$, there exists a non-closed implication tree of $(C, \mathcal{G}(D))$. In order to prove the existence of a UML-model of D, a canonical interpretation is constructed by consistent subtrees of the non-closed implication trees of $(C_1, \mathcal{G}(D)), \ldots, (C_n, \mathcal{G}(D))$ (with $cls(\mathcal{G}(D)) = \{C_1, \ldots, C_n\}$) and by $Assoc(\mathcal{G}(D)) = 1$. This proves that D is consistent.

Corresponding to calling $Cons(\delta_0, \emptyset, \Gamma_0)$, we define an implication tree of a class set δ_0 that expresses the consistency checking proof of δ_0.

Definition 4. *Let Γ_0 be a set of implication forms and let $\delta_0 \subseteq cls(\Gamma_0)$. An implication tree of (δ_0, Γ_0) is a finite and minimal tree such that (i) the root is a node labeled with δ_0, (ii) each non-leaf node is labeled with a non-empty set of classes, (iii) each leaf is labeled with 0, 1, or w, (iv) each edge is labeled with Γ or (Γ, a) where $\Gamma \in \Sigma(\Gamma_0)$ and $a \in att(\Gamma_0)$, and (v) for each node labeled with δ and each $\Gamma \in \Sigma(\Gamma_0)$, if $\bigcup_{C \in \delta} H(C, \Gamma)$ contains $\{C, \neg C\}$ or $\{t_1, \ldots, t_n\}$ with $t_1 \cap \cdots \cap t_n = \emptyset$, then there is a child of δ labeled with 0 and the edge of the nodes δ and 0 is labeled with Γ, and otherwise:*

- *if $att(\Gamma_0) = \emptyset$, then there is a child of δ labeled with 1 and the edge of the nodes δ and 1 is labeled with Γ, and*
- *for all $a \in att(\Gamma_0)$, the following conditions hold:*
  1. *if $i > j$ such that $\{\geq i, \leq j\} \in N(\delta, a, \Gamma)$, then there is a child of δ labeled with 0 and the edge of the nodes δ and 0 is labeled with (Γ, a),*
  2. *if $E(\delta, a, \Gamma) = \emptyset$, then there is a child of δ labeled with 1 and the edge of the nodes δ and 1 is labeled with (Γ, a),*
  3. *if there is an ancestor labeled with $E(\delta, a, \Gamma)$ or $\mu_0(E(\delta, a, \Gamma), \Gamma)$, then there is a child of δ labeled with w and the edge of the nodes δ and w is labeled with (Γ, a), and*
  4. *otherwise, there is a child of δ labeled with $E(\delta, a, \Gamma)$ and the edge of the nodes δ and $E(\delta, a, \Gamma)$ is labeled with (Γ, a).*

Let $\mathcal{T}$ be an implication tree of (δ_0, Γ_0). A node d in $\mathcal{T}$ is closed if (i) d is labeled with 0 or if (ii) d is labeled with δ and for every $\Gamma \in \Sigma(\Gamma_0)$, there is an edge (d, d') labeled with Γ or (Γ, a) such that d' is closed. An implication tree of

(δ_0, Γ_0) is closed if the root is closed; it is non-closed otherwise. A forest of Γ_0 is a set of implication trees of $(\{C_1\}, \Gamma_0), \ldots, (\{C_n\}, \Gamma_0)$ such that $cls(\Gamma_0) = \{C_1, \ldots, C_n\}$. A forest $\mathcal{S}$ of Γ_0 is closed if there exists a closed implication tree $\mathcal{T}$ in $\mathcal{S}$. The following lemma states the correspondence between the consistency checking for every $C \in cls(\Gamma_0)$ and the existence of a non-closed forest of Γ_0.

Lemma 1. *Let Γ_0 be a set of implication forms. For every class $C \in cls(\Gamma_0)$, $Cons(\{C\}, \emptyset, \Gamma_0) = 1$ if and only if there is a non-closed forest of Γ_0.*

We define a consistent subtree $\mathcal{T}'$ of a non-closed implication tree $\mathcal{T}$ such that $\mathcal{T}'$ is constructed by non-closed nodes in $\mathcal{T}$.

Definition 5 (Consistent Subtree). *Let $\mathcal{T}$ be a non-closed implication tree of $(\{C_0\}, \Gamma_0)$ and d_0 be the root where Γ_0 is a set of implication forms and $C_0 \in cls(\Gamma_0)$. A tree $\mathcal{T}'$ is a consistent subtree of $\mathcal{T}$ if (i) $\mathcal{T}'$ is a subtree of $\mathcal{T}$, (ii) every node in $\mathcal{T}'$ is not closed, and (iii) every non-leaf node has m children of all the attributes $a_1, \ldots, a_m \in att(\Gamma_0)$ where each child is labeled with 1, w, or a set of classes and each edge of the non-leaf node and its child is labeled with (Γ, a_i).*

We show the correspondence between the consistency of an implication form set Γ_0 and the existence of a non-closed forest of Γ_0. We extend the first-order language by adding the new constants $\bar{d}$ for all the elements $d \in U$ such that each new constant is interpreted by itself, i.e., $I(\bar{d}) = d$. In addition, we define the following operations:

1. $proj^n_k(x_1, \ldots, x_n) = x_k$ where $1 \leq k \leq n$.
2. $Max_{\geq}(X) = (Max(X_1), \ldots, Max(X_n))$ where X is a set of n-tuples and for each $v \in \{1, \ldots, n\}$, $X_v = \{\ proj^n_v\ (i_1, \ldots, i_n) \mid (\geq i_1, \ldots, \geq i_n) \in X\}$.
3. $AC(A, \Gamma) = (C_1, \ldots, C_n)$ if $A(x_1, \ldots, x_n) \rightarrow C_1(x_1) \wedge \cdots \wedge C_n(x_n) \in \Gamma$.

A canonical interpretation of an implication form set Γ_0 is constructed by consistent subtrees of the non-closed implication trees in a forest of Γ_0, that is used to prove the completeness of the algorithm $Cons$. A class C is consistent in Γ if there exists a non-closed implication tree of $(\{C\}, \Gamma_0)$ such that the root labeled with $\{C\}$ has a non-closed child node labeled with Γ or (Γ, a).

Definition 6 (Canonical Interpretation). *Let Γ_0 be a set of implication forms such that $Assoc(\Gamma_0) = 1$ and let $\mathcal{S} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ be a non-closed forest of Γ_0. For every $\mathcal{T}_i \in \mathcal{S}$, there is a consistent subtree $\mathcal{T}'_i$ of $\mathcal{T}_i$, and we set $\mathcal{S}' = \{\mathcal{T}'_1, \ldots, \mathcal{T}'_n\}$ as the set of consistent subtrees of $\mathcal{T}_1, \ldots, \mathcal{T}_n$ in $\mathcal{S}$. An canonical interpretation of Γ_0 is a pair $\mathcal{I} = (U, I)$ such that $U_0 = \{d \mid d$ is a non-leaf node in $\mathcal{T}'_1 \cup \cdots \cup \mathcal{T}'_n\}$, each $e_0, e_j, e_{(v,w)}$ are new individuals, and the following conditions hold:*

1. $U = U_0 \cup \bigcup_{\substack{d \in \mathcal{T}'_1 \cup \cdots \cup \mathcal{T}'_n \\ a \in att(\Gamma_0)}} U_{d,a} \cup \bigcup_{\substack{d \in \mathcal{T}'_1 \cup \cdots \cup \mathcal{T}'_n \\ A \in asc(\Gamma_0)}} U_{d,A}$ *and* $I(x) = I_0(x) \cup \bigcup_{\substack{d \in \mathcal{T}'_1 \cup \cdots \cup \mathcal{T}'_n \\ a \in att(\Gamma_0)}} I_{d,a}(x) \cup \bigcup_{\substack{d \in \mathcal{T}'_1 \cup \cdots \cup \mathcal{T}'_n \\ A \in asc(\Gamma_0)}} I_{d,A}(x)$.

2. For each $\Gamma \in \Sigma(\Gamma_0)$,

– $d \in I_0(C)$ *iff a non-leaf node* d *is labeled with* δ *where* $C \in \bigcup_{C' \in \delta} H(C', \Gamma)$, *and*
– $(d, d') \in I_0(a)$ *iff (i)* d' *is a non-leaf node and* (d, d') *is an edge labeled with* (Γ, a), *or (ii) a node* d *has a child labeled with* w, *there is a witness* d_0 *of* d, *and* (d_0, d') *is an edge labeled with* (Γ, a).

3. For each edge (d, d') *labeled with* (Γ, a) *such that the node* d *is labeled with* δ *and* $Max_{\geq}(N(\delta, a, \Gamma)) = k$,

– $U_{d,a} = \{e_1, \ldots, e_{k-1}\}$,
– $(d, e_1), \ldots, (d, e_{k-1}) \in I_{d,a}(a)$ *iff* $(d, d') \in I_0(a)$,
– $e_1, \ldots, e_{k-1} \in I_{d,a}(C)$ *iff* $d' \in I_0(C)$, *and*
– $(e_1, d''), \ldots, (e_{k-1}, d'') \in I_{d,a}(a')$ *iff* $(d', d'') \in I_0(a')$.

4. For all nodes $d \in I_0(C_k)$ *such that* $AC(A, \Gamma_0) = (C_1, \ldots, C_k, \ldots, C_n)$ *and* $Max_{\geq}(N_k(H(A, \Gamma_0), \Gamma_0)) = (i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n)$,

– $U_{d,A} = \{e_0\} \cup \bigcup_{v \in \{1,\ldots,n\} \setminus \{k\}} \{e_{(v,1)}, \ldots, e_{(v,i_v)}\}$,
– *for all* $(w_1, \ldots, w_{k-1}, w_{k+1}, \ldots, w_n) \in \mathbf{N}^n$ *such that* $1 \leq w_v \leq i_v$, $(e_{(1,w_1)}, \ldots, e_{(k-1,w_{k-1})}, d, e_{(k+1,w_{k+1})}, \ldots, e_{(n,w_n)}) \in I_{d,A}(A)$ *and* $e_{(1,w_1)} \in I_{d,A}(C_1)$, $\ldots$, $e_{(k-1,w_{k-1})} \in I_{d,A}(C_{k-1})$, $e_{(k+1,w_{k+1})} \in I_{d,A}(C_{k+1})$, $\ldots$, $e_{(n,w_n)} \in I_{d,A}(C_n)$,
– $e_{(v,w)} \in I_{d,A}(C')$ *for all* $C' \in H(C_v, \Gamma')$ *iff* $e_{(v,w)} \in I_{d,A}(C_v)$ *and* C_v *is consistent in* Γ',
– $(u_1, \ldots, u_n) \in I_{d,A}(A')$ *for all* $A' \in H(A, \Gamma_0)$ *iff* $(u_1, \ldots, u_n) \in I_{d,A}(A)$[3],
– $(e_{(v,w)}, d'') \in I_{d,A}(a)$ *and* $e_{(v,w)} \in I_{d,A}(C_v)$ *iff* $(d', d'') \in I_0(a)$ *and* $d' \in I_0(C_v)$, *and*
– *for all* $(w_1, \ldots, w_{k-1}, w_{k+1}, \ldots, w_n) \in \mathbf{N}^n$ *such that* $1 \leq w_v \leq i_v$, *and* $(e_{(1,w_1)}, \ldots, e_{(k-1,w_{k-1})}, e, e_{(k+1,w_{k+1})}, \ldots, e_{(n,w_n)}) \in I_{d,A}(A)$ *iff* $e \in I(C_k)$ *where* e *is* e_0, e_j, *or* $e_{(x,y)}$.

5. For all $A \in asc(\Gamma_0)$,

– $(u_1, \ldots, u_n, e_0) \in I_{d,A}(r_0)$ *and* $e_0 \in I_{d,A}(C_A)$ *iff* $(u_1, \ldots, u_n) \in I_{d,A}(A)$,
– $e_0 \in I_{d,A}(C)$ *for all* $C \in H(C_A, \Gamma')$ *iff* $e_0 \in I_{d,A}(C_A)$ *and* C_v *is consistent in* Γ', *and*
– $(e_0, d'') \in I_{d,A}(a)$ *and* $e_0 \in I_{d,A}(C_A)$ *iff* $(d', d'') \in I_0(a)$ *and* $d' \in I_0(C_A)$.

Lemma 2. *Let* Γ_0 *be a set of implication forms. There exists an interpretation* $\mathcal{I}$ *such that for every* $C_0 \in cls(\Gamma_0)$, $\mathcal{I} \models \exists x.C_0(x)$ *if and only if (i) there exists a non-closed forest of* Γ_0 *and (ii)* $Assoc(\Gamma_0) = 1$.

The correctness for the algorithms *Cons* and *Assoc* is obtained as follows:

Theorem 1 (Soundness and completeness). *Let* D *be a UML class diagram with association generalization and without roles, and let* $\mathcal{G}(D)$ *be the translation of* D *into a set of implication forms.* D *is consistent if and only if* $Cons(\{C\}, \emptyset, \mathcal{G}(D)) = 1$ *for all* $C \in cls(\mathcal{G}(D))$ *and* $Assoc(\mathcal{G}(D)) = 1$.

Theorem 2 (Termination). *The consistency checking algorithm Cons terminates.*

[3] Note that d, d', d'', d_0 are nodes, $e_0, e_j, e_{(v,w)}$ are new constants, and u, u_j are nodes or new constants.

5 Algorithms and Complexities for Various Expressivities

The proposed consistency checking algorithm terminates; however, *Cons* still exhibits a double-exponential complexity in the worst case (and *Assoc* exhibits polynomial time complexity). In this section, we will present optimized consistency checking algorithms for class diagrams of different expressive powers.

5.1 Restriction of Inconsistency Triggers

We denote the set of UML class diagrams with association generalization and without roles as $\mathcal{D}^-_{ful}$. By deleting certain inconsistency triggers, we classify UML class diagrams that are less expressive than $\mathcal{D}^-_{ful}$. The least set $\mathcal{D}^-_0$ of class diagrams is obtained by deleting disjointness/completeness constraints and overwriting/multiple inheritances. We define $\mathcal{D}^-_{dis}$, $\mathcal{D}^-_{com}$, and $\mathcal{D}^-_{inh}$ as extensions of $\mathcal{D}^-_0$ by adding disjointness constraints, completeness constraints, and overwriting/multiple inheritances, respectively. We denote $\mathcal{D}^-_{dis+com}$, $\mathcal{D}^-_{dis+inh}$, and $\mathcal{D}^-_{inh+com}$ as the unions of $\mathcal{D}^-_{dis}$ and $\mathcal{D}^-_{com}$, $\mathcal{D}^-_{dis}$ and $\mathcal{D}^-_{inh}$, and $\mathcal{D}^-_{inh}$ and $\mathcal{D}^-_{com}$, respectively. In order to design algorithms suitable for these expressivities, we divide the class diagrams into the five groups, as shown in Fig.2.

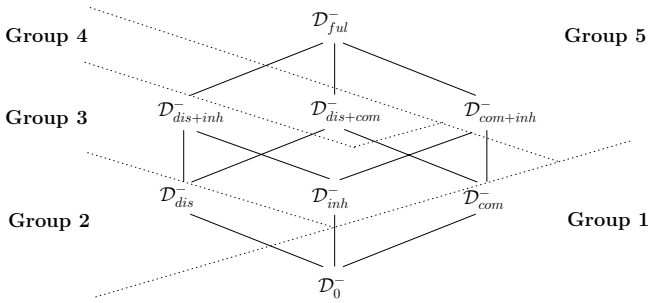


Fig. 2. Classification of UML class diagrams

The least expressive Group 1 is the set of class diagrams obtained by deleting disjointness constraints and overwriting/multiple inheritances (but allowing attribute multiplicities). Groups 2 and 3 prohibit $C_1(x) \vee \cdots \vee C_m(x)$ as disjunctive classes by deleting completeness constraints, and furthermore, Group 2 contains no overwriting/multiple inheritances. Group 4 is restricted by eliminating overwriting/multiple inheritances (but allowing disjointness constraints, completeness constraints, and attribute multiplicities).

5.2 Restriction of Attribute Value Types

Apart from the restriction of inconsistency triggers, we naturally restrict attribute value types in the overwriting/multiple inheritances. Consider the class

hierarchy in Fig.3. Class C_1 with attribute $a\colon C$ inherits attributes $a\colon C'$ and

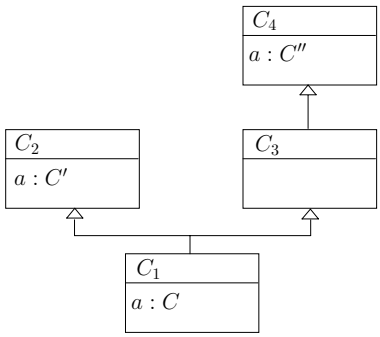


Fig. 3. Attribute value types in overwriting/multiple inheritances

$a\colon C''$ from superclasses C_2 and C_4. In this case, if the value type C is a subclass of all the other value types C' and C'' of the same name's attributes in the class hierarchy, then the consistency checking of the value types C, C', and C'' can be guaranteed by the consistency checking of only the value type C.

Let $C \in cls(\Gamma_0)$ and $\Gamma \in \Sigma(\Gamma_0)$. The value types of attributes in class C are said to be *restrictedly defined in* Γ when if the superclasses $C_1, \ldots, C_n$ of C (i.e., $H(C, \Gamma) = \{C_1, \ldots, C_n\}$) have the same name's attributes and the value types are classes $C'_1, \ldots, C'_m$, then a value type C'_i is a subclass of the other value types $\{C'_1, \ldots, C'_m\} \backslash \{C'_i\}$ (i.e.,$\{C'_1, \ldots, C'_m\} \subseteq H(C'_i, \Gamma)$). Every attribute value type is restrictedly defined if the value types of attributes in any class $C \in cls(\Gamma_0)$ are restrictedly defined in any $\Gamma \in \Sigma(\Gamma_0)$. For example, as shown in Fig.3, the value types C, C', and C'' of attribute a in class C_1 are restrictedly defined in $\Gamma_1 = \{C_1(x) \to C_2(x)$, $C_1(x) \to C_3(x)$, $C_3(x) \to C_4(x)$, $C_1(x) \to (a(x, y) \to C(y))$, $C_2(x) \to (a(x, y) \to C'(y))$, $C_4(x) \to (a(x, y) \to C''(y)), \ldots\}$ if $\{C, C', C''\} \subseteq H(C_0, \Gamma_1)$, where C_0 is C, C', or C''.

5.3 Optimized Algorithms

We show that Group 1 does not cause any inconsistency and we devise consistency checking algorithms suitable for Groups 2–5. The following algorithm $Cons1$ computes the consistency of class diagrams in $\mathcal{D}^-_{dis+inh}$, $\mathcal{D}^-_{inh}$, and $\mathcal{D}^-_{dis}$ if we call $Cons1(\{C_0\}, \emptyset, \Gamma_0)$ for every class $C_0 \in cls(\Gamma_0)$. Let X be a set and Y be a family of sets. Then, we define $ADD(X, Y) = \{X_i \in Y \mid X_i \not\subset X\} \cup \{X\}$ such that X is added to Y and all $X_i \subset X$ are removed from Y. Since $\mathcal{D}^-_{dis+inh}$, $\mathcal{D}^-_{inh}$, and $\mathcal{D}^-_{dis}$ do not contain any completeness constraints, there is a unique decomposed set of Γ_0, namely, $\Sigma(\Gamma_0) = \{\Gamma\}$. Instead of recursive calls, $Cons1$ performs looping of consistency checking for each element of variable P that stores unchecked sets of classes.

Algorithm $Cons1$ for $\mathcal{D}^{-}_{dis+inh}$, $\mathcal{D}^{-}_{inh}$, and $\mathcal{D}^{-}_{dis}$
input set of classes δ, family of sets of classes Δ, set of implication forms Γ_0
output 1 (consistent) or 0 (inconsistent)
begin
  $P = \{\delta\}$; $G = \Delta$;
  while $P \neq \emptyset$ **do**
    $\delta \in P$; $P = P - \{\delta\}$; $\Gamma \in \Sigma(\Gamma_0)$; $S = \bigcup_{C \in \delta} H(C, \Gamma)$;
    if $\{C, \neg C\} \subseteq S$ or $\{t_1, \ldots, t_n\} \subseteq S$ s.t. $t_1 \cap \cdots \cap t_n = \emptyset$ **then return** 0;
    else $G = ADD(\delta, G)$;
      for $a \in att(\Gamma_0)$ **do**
        if $i > j$ s.t. $\{\geq i, \leq j\} \subseteq N(\delta, a, \Gamma)$ **then return** 0;
        else $\delta_a = E(\delta, a, \Gamma)$;
          if $\delta_a \neq \emptyset$ and $\delta_a, \mu_0(\delta_a, \Gamma) \not\subseteq \delta'$ for all $\delta' \in G$ **then**
            if $\mu(\delta_a, \Gamma) \neq \emptyset$ **then** $\delta_a = \mu_0(\delta_a, \Gamma)$;
            $P = ADD(\delta_a, P)$;
          fi
        esle
      rof
    esle
  elihw
  return 1;
end;

The algorithm $Cons2$ is simply designed for testing the consistency of an input class C_0 in every $\Gamma \in \Sigma(\Gamma_0)$, where the multiplicities of attributes in C_0 are checked but the disjointness of its attribute value types are not. This is because $\mathcal{D}^{-}_{dis+com}$ involves no overwriting/multiple inheritances, i.e., each attribute value is uniquely typed and if type T is a class (in $cls(\Gamma_0)$), the consistency of T can be checked in another call $Cons2(T, \Gamma_0)$. This algorithm computes the consistency of $\mathcal{D}^{-}_{dis+com}$ if $Cons2(C_0, \Gamma_0)$ is called for every class $C_0 \in cls(\Gamma_0)$.

Algorithm $Cons2$ for $\mathcal{D}^{-}_{dis+com}$
input class C_0, set of implication forms Γ_0
output 1 (consistent) or 0 (inconsistent)
begin
  for $\Gamma \in \Sigma(\Gamma_0)$ **do**
    $S = H(C_0, \Gamma)$;
    if $\{C, \neg C\} \not\subseteq S$ and $\{t_1, \ldots, t_n\} \not\subseteq S$ s.t. $t_1 \cap \cdots \cap t_n = \emptyset$ **then**
      for $a \in att(\Gamma_0)$ **do**
        if $i > j$ s.t. $\{\geq i, \leq j\} \subseteq N(C_0, a, \Gamma)$ **then return** 0;
      return 1;
    fi
  rof
  return 0;
end;

It should be noted that the algorithm $Cons$ requires double exponential time in the worst case. We develop the optimized algorithm $Cons3$ as the single exponential version by skipping the sets of classes that are already checked as

consistent or inconsistent in any former routine (but $Cons$ limits the skipping to the set Δ stored in the caller processes). It computes the consistency of $\mathcal{D}^-_{com+inh}$ and $\mathcal{D}^-_{ful}$ if we call $Cons3(\{C_0\}, \emptyset, \Gamma_0)$ for every class $C_0 \in cls(\Gamma_0)$.

Algorithm $Cons3$ for $\mathcal{D}^-_{com+inh}$ and $\mathcal{D}^-_{ful}$
input set of classes δ, family of sets of classes Δ, set of implication forms Γ_0
output 1 (consistent) or 0 (inconsistent)
global variables $G = \emptyset$, $NG = \emptyset$
begin
  for $\Gamma \in \Sigma(\Gamma_0)$ s.t. $(\delta, \Gamma) \notin NG$ **do**
    $S = \bigcup_{C \in \delta} H(C, \Gamma)$; $f_\Gamma = 0$;
    if $\{C, \neg C\} \not\subseteq S$ and $\{t_1, \ldots, t_n\} \not\subseteq S$ s.t. $t_1 \cap \cdots \cap t_n = \emptyset$ **then** $f_\Gamma = 1$;
      for $a \in att(\Gamma_0)$ **do**
        if $i > j$ s.t. $\{\geq i, \leq j\} \subseteq N(\delta, a, \Gamma)$ **then** $f_\Gamma = 0$;
        else $\delta_a = E(\delta, a, \Gamma)$;
          if $\delta_a \neq \emptyset$ and $\delta_a, \mu_0(\delta_a, \Gamma) \not\subseteq \delta'$ for all $\delta' \in \Delta \cup G$ **then**
            if $\mu(\delta_a, \Gamma) \neq \emptyset$ **then** $\delta_a = \mu_0(\delta_a, \Gamma)$;
            $f_\Gamma = Cons3(\delta_a, \Delta, \Gamma_0)$;
          fi
        esle
      rof
    fi
    if $f_\Gamma = 1$ **then** $G = ADD(\delta, G)$; **return** 1;
    else $NG = ADD((\delta, \Gamma), NG)$;
  rof
  return 0;
end;

The optimization method of using good and no good variables G and NG is based on the EXPTIME tableau algorithm in [4]. In $Cons1$ and $Cons3$, the good variable $G = \{\delta_1, \ldots, \delta_n\}$ contains sets of classes such that each set δ_i is consistent in a decomposed set of Γ_0 (in $\Sigma(\Gamma_0)$). In $Cons3$, the no good variable NG contains pairs of a set δ of classes and a decomposed set Γ of Γ_0 such that δ is inconsistent in Γ. Each element in NG exactly indicates the inconsistency of δ in the set Γ by storing the pair (δ, Γ), so that it is never checked again. In addition to this method, we consider that further elements can be skipped by the condition "$\delta_a, \mu_0(\delta_a, \Gamma) \not\subseteq \delta'$ for all $\delta' \in \Delta \cup G$." This implies that $Cons1$ and $Cons3$ skip the consistency checking of the target set δ_a if a superset δ' of either δ_a or $\mu_0(\delta_a, a, \Gamma)$ is already checked in former processes (i.e., $\delta' \in \Delta \cup G$). With regard to the skipping condition, the following lemma guarantees that if $\mu(\delta, \Gamma) \neq \emptyset$, then all the classes $C_1, \ldots, C_n$ in δ and the sole class C in $\mu_0(\delta, \Gamma)$ $(= \{C\})$ have the same superclasses. In other words, the consistency checking of δ can be replaced with the consistency checking of $\mu_0(\delta, \Gamma)$. Therefore, the computational steps can be decreased by skipping the target set δ_a since this set can be replaced by an already checked superset of the singleton $\mu_0(\delta_a, a, \Gamma)$.

Lemma 3. *Let Γ_0 be a set of implication forms and let $\Gamma \in \Sigma(\Gamma_0)$. For all $\delta \subseteq cls(\Gamma_0)$ and $a \in att(\Gamma_0)$, if $\mu(\delta, \Gamma) \neq \emptyset$, then $\bigcup_{C \in \delta} H(C, \Gamma) = \bigcup_{C \in \mu_0(\delta, \Gamma)} H(C, \Gamma)$, $N(\delta, a, \Gamma) = N(\mu_0(\delta, \Gamma), a, \Gamma)$, and $E(\delta, a, \Gamma) = E(\mu_0(\delta, \Gamma), a, \Gamma)$.*

We adjust the algorithm $Cons3$ to class diagrams in which every attribute value type is restrictedly defined. The algorithm $Cons4$ is shown below; as indicated by the underlined text, this algorithm is improved by only storing sets of classes in NG (similar to G). The restriction of value types leads to $\mu(\delta_a, \Gamma) \neq \emptyset$; therefore, the size of NG is limited to a set of singletons of classes. In other words, $Cons4$ can be adjusted to decrease the space complexity (i.e., NG) to polynomial space by using the property of Lemma 3. Unfortunately, this adjustment does not yield a single exponential algorithm if attribute value types are *unrestrictedly* defined. Hence, we need both $Cons3$ and $Cons4$ for the case where attribute value types are restrictedly defined or not.

Algorithm $Cons4$ for $\mathcal{D}^-_{com+inh}$ and $\mathcal{D}^-_{ful}$
input set of classes δ, family of sets of classes Δ, set of implication forms Γ_0
output 1 (consistent) or 0 (inconsistent)
global variables $G = \emptyset$, $NG = \emptyset$
begin
  for $\Gamma \in \Sigma(\Gamma_0)$ **do**
    $S = \bigcup_{C \in \delta} H(C, \Gamma)$; $f_\Gamma = 0$;
    if $\{C, \neg C\} \not\subseteq S$ and $\{t_1, \ldots, t_n\} \not\subseteq S$ s.t. $t_1 \cap \cdots \cap t_n = \emptyset$ **then** $f_\Gamma = 1$;
      for $a \in att(\Gamma_0)$ **do**
        if $i > j$ s.t. $\{\geq i, \leq j\} \subseteq N(\delta, a, \Gamma)$ **then** $f_\Gamma = 0$;
        else $\delta_a = E(\delta, a, \Gamma)$;
          if $\delta_a \neq \emptyset$ and $\delta_a, \mu_0(\delta_a, \Gamma) \not\subseteq \delta'$ for all $\delta' \in \Delta \cup G$ **then**
            if $\mu(\delta_a, \Gamma) \neq \emptyset$ **then** $\delta_a = \mu_0(\delta_a, \Gamma)$;
            if $\delta_a \in NG$ **then** $f_\Gamma = 0$;
            else $f_\Gamma = Cons4(\delta_a, \Delta, \Gamma_0)$;
          fi
        esle
      rof
    fi
    if $f_\Gamma = 1$ **then** $G = ADD(\delta, G)$; **return** 1;
  rof
  $NG = ADD(\delta, NG)$; **return** 0;
end;

Without losing the completeness of consistency checking (see Appendix in [8]), these algorithms have the following computational properties for each class diagram group (as shown in Table 1). We believe that the complexity classes 0, P, NP, and PSPACE less than EXPTIME are suitable for us to implement the algorithms for different expressive powers of class diagram groups. For all the class diagram groups, complexity1 in Table 1 arranges the complexities of algorithms $Cons1$, $Cons2$, and $Cons3$ with respect to the size of a class diagram. Every class diagram in $\mathcal{D}^-_0$ and $\mathcal{D}^-_{com}$ is consistent; therefore, the complexity is zero (i.e., we do not need to check consistency). $Cons1$ computes the consistency of $\mathcal{D}^-_{dis}$ in P (polynomial time) and that of $\mathcal{D}^-_{inh}$ and $\mathcal{D}^-_{dis+inh}$ in EXPTIME (exponential time). $Cons2$ computes the consistency of $\mathcal{D}^-_{dis+com}$ in NP (non-deterministic polynomial time), and $Cons3$ computes the consistency of $\mathcal{D}^-_{com+inh}$ and $\mathcal{D}^-_{ful}$ in EXPTIME.

Moreover, complexity2 in Table 1 shows the complexities of the algorithms $Cons1$, $Cons2$, and $Cons4$ for the case in which every attribute value type is restrictedly defined. In particular, $Cons1$ computes the consistency of $\mathcal{D}^-_{inh}$ and $\mathcal{D}^-_{dis+inh}$ in P, and $Cons4$ computes the consistency of $\mathcal{D}^-_{com+inh}$ and $\mathcal{D}^-_{ful}$ in PSPACE (polynomial space). Therefore, by Lemma 3 and by the skipping method of consistency checking, the complexities of $Cons1$ and $Cons4$ are respectively reduced from EXPTIME to P and PSPACE. Due to spatial constraints, detailed proofs of the lemmas and theorems have been omitted (see [8]).

6 Conclusion

We introduced the restriction of UML class diagrams based on

(i) inconsistency triggers (disjointness constraints, completeness constraints, and overwriting/multiple inheritances) and
(ii) attribute value types defined with restrictions in overwriting/multiple inheritances.

Inconsistency triggers are employed to classify the expressivity of class diagrams, and their combination with the attribute value types results in tractable consistency checking of the restricted class diagrams. First, we presented a complete algorithm for testing the consistency of class diagrams including any inconsistency triggers. Second, the algorithm was suitably refined in order to develop optimized algorithms for different expressive powers of class diagrams obtained by deleting some inconsistency triggers. Our algorithms were easily modified depending on the presence of diagram components. The algorithms clarified that every class diagram in $\mathcal{D}^-_0$ and $\mathcal{D}^-_{com}$ must have a UML-model (i.e., consistency is guaranteed) and when every attribute value type is restrictedly defined, the complexities of class diagrams in $\mathcal{D}^-_{inh}$ and $\mathcal{D}^-_{dis+inh}$ and in $\mathcal{D}^-_{com+inh}$ and $\mathcal{D}^-_{ful}$ are essentially decreased from EXPTIME to P and PSPACE, respectively. Restricted/classified UML class diagrams and their optimized algorithms are new results; however, the translation into first-order logic is similar to and based on the study of [1, 2].

Our future research is concerned with the complexity in terms of the depth of class hierarchies and the average-case complexity for consistency checking. Furthermore, an experimental evaluation should be performed in order to ascertain the applicability of optimized consistency algorithms.

References

1. B. Beckert, U. Keller, and P. H. Schmitt. Translating the object constraint language into first-order predicate logic. In *Proceedings of VERIFY, Workshop at Federated Logic Conferences (FLoC)*, 2002.
2. D. Berardi, A. Cali, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2):70–118, 2005.

Table 1. Upper-bound complexities of algorithms for testing consistency

UML group	complexity1	algorithm	complexity2	algorithm
$\mathcal{D}_0^-$	**0**		**0**	
$\mathcal{D}_{com}^-$	**0**		**0**	
$\mathcal{D}_{dis}^-$	**P**	$Cons1$	**P**	$Cons1$
$\mathcal{D}_{inh}^-$	EXPTIME		**P**	
$\mathcal{D}_{dis+inh}^-$	EXPTIME		**P**	
$\mathcal{D}_{dis+com}^-$	**NP**	$Cons2$	**NP**	$Cons2$
$\mathcal{D}_{com+inh}^-$	EXPTIME	$Cons3$	**PSPACE**	$Cons4$
$\mathcal{D}_{ful}^-$	EXPTIME		**PSPACE**	

3. F. M. Donini. Complexity of reasoning. In *Description Logic Handbook*, pages 96–136, 2003.
4. F. M. Donini and F. Massacci. EXPTIME tableaux for ALC. *Artificial Intelligence*, 124(1):87–138, 2000.
5. A. S. Evans. Reasoning with UML class diagrams. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98, USA*, 1998.
6. M. Fowler. *UML Distilled: A Brief Guide to the Standard Modeling Object Language*. Object Technology Series. Addison-Wesley, third edition, September 2003.
7. E. Franconi and G. Ng. The i.com tool for intelligent conceptual modeling. In *KRDB*, pages 45–53, 2000.
8. K. Kaneiwa and K. Satoh. Consistency checking algorithms for restricted UML class diagrams. NII Technical Report, NII-2005-013E, National Institute of Informatics, 2005, http://research.nii.ac.jp/TechReports/05-013E.html.
9. P. G. Kolaitis and J. A. Väänänen. Generalized quantifiers and pebble games on finite structures. *Annals of Pure and Applied Logic*, 74(1):23–75, 1995.
10. H. Mannila and K.-J. Räihä. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40(2):237–243, 1992.
11. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1st edition, 1999.
12. K.-D. Schewe and B. Thalheim. Fundamental concepts of object oriented databases. *Acta Cybern*, 11(1-2):49–84, 1993.
13. A. Tsiolakis and H. Ehrig. Consistency analysis between UML class and sequence diagrams using attributed graph gammars. In *Proceedings of joint APPLIGRAPH/ GETGRATS Workshop on Graph Transformation Systems*, pages 77–86, 2000.