Reasoning with UML Class Diagrams: Relevance, Problems, and Solutions – a Survey

Mira Balaban¹, Azzam Maraee², Arnon Sturm²

¹Department of Computer Science, ²Department of Information Systems Engineering,
Ben-Gurion University of the Negev, Beer Sheva 84105, Israel
mira@cs.bgu.ac.il, mari@bgu.ac.il, sturm@bgu.ac.il

Abstract
UML is now widely accepted as the standard modeling language for software construction. The Class Diagram model is its core view, providing the backbone for any modeling effort and having well formed semantics. Class diagrams are widely used for purposes such as software and language specification, database and ontology engineering, and model transformation. The central role played by class diagrams emphasizes the need for strengthening Computer Aided Software Engineering (CASE) tools with features at the level of modern integrated development environments. Such tools need to detect errors in models, identify the source of errors, reveal redundancies, and possibly suggest design improvements. These tasks require reasoning capabilities since they depend on the meaning of the constraints that occur in a diagram.

This paper characterizes four essential reasoning problems in class diagram design, and surveys methods for their solution. The contribution of this paper lies in classifying reasoning problems in the class diagram model, presenting the state of the art results for these problems, and enlightening key issues for future research.

Keywords: Reasoning, Conceptual Modeling, UML, class diagrams, consistency, finite satisfiability, redundancy, design quality, inconsistency detection.
Reasoning with UML Class Diagrams: Problems, Relevance and Efficiency – a Survey

Abstract

UML is now widely accepted as the standard modeling language for software construction. The Class Diagram model is its core view, providing the backbone for any modeling effort and having well formed semantics. Class diagrams are widely used for purposes such as software and language specification, database and ontology engineering, and model transformation. The central role played by class diagrams emphasizes the need for strengthening Computer Aided Software Engineering (CASE) tools with features at the level of modern integrated development environments. Such tools need to detect errors in models, identify the source of errors, reveal redundancies, and possibly suggest design improvements. These tasks require reasoning capabilities since they depend on the meaning of the constraints that occur in a diagram.

This paper characterizes four essential reasoning problems in class diagram design, and surveys methods for their solution. The contribution of this paper lies in classifying reasoning problems in the class diagram model, presenting the state of the art results for these problems, and enlightening key issues for future research.

1 Introduction

The Unified Modeling Language (UML) is the standard de facto for system development as it was developed and adopted by the Object Management Group [68]. It consists of several languages, each describing a different view of object-oriented software. The most important view of UML is the static/structural specification that is expressed as a class diagram. A class diagram
is a structural abstraction of the real world which consists of classes and their descriptors, associations among them, and constraints imposed on both classes and associations.

The origin of the class diagram model is the conceptual models of the 80's, like Entity-relationship (ER) diagrams [23], their Enhanced versions (EER), Object-Role Modeling (ORM) diagrams [42], and Frames structured modeling in artificial intelligence [67]. The UML class diagram model includes elements from all of these models.

Among the nine visual UML models, class diagrams appear to be the most clear, intuitive and well defined. Dobing and Parsons performed a survey on the usage of UML [28]. In that survey, the class diagram was most frequently used (73 percents). It was found useful in clarifying technical understanding, for programmer specification, and for maintenance documentation. It was found less useful for client verification. The major usage of UML class diagrams is to specify, visualize, and document the system static view. They also serve as a basis for generating implementation artifacts such as code skeleton and database schemata, as a means for knowledge representation such as specifying ontologies, and for defining meta-models of other programming, modeling, and specification languages.

The central role that class diagrams play in the design and specification of software, databases and ontologies emphasizes the need for powerful Computer Aided Software Engineering (CASE) tools, at the level of current Integrated Development Environments (IDE). Such tools need to prevent syntax errors, provide model critique, and possibly suggest design improvements. Model critique can detect redundancies and contradictions, and identify the reasons for errors. Such tasks require reasoning capabilities since they depend on the meaning of the constraints that occur in a diagram. Furthermore, reasoning enables better model management as it provides capabilities for querying the model.
In this paper, we characterize four essential problems that require reasoning on class diagrams, and survey main research results in coping with these problems. The contribution of the paper lies in classifying reasoning problems with respect to the class diagram model, and presenting the state of the art results for these problems, while enlightening the key issues for future research.

The structure of the rest of the paper is as follows. Section 2 introduces the class diagram notions, and discusses the relevance of applying reasoning techniques when analyzing it. Section 3 presents the classification of class diagram reasoning problems, and the survey of available solutions. Section 4 concludes and draws the line for future research.

2. The UML Class Diagram

A class diagram is a structural abstraction of a real world phenomenon. The model consists of basic elements, descriptors and constraints. The basic elements are classes and associations, the descriptors are class and association attributes, and the constraints are restrictions imposed on these elements. The constraints are (1) cardinality constraints on associations, with or without qualifiers; (2) association class constraint; (3) class and associations hierarchy constraints; (4) generalization set constraints; (5) association constraints; (6) aggregation constraints; and (7) cardinality constraints on attributes. The syntax and informal semantics are described in [68, 74].

Figure 1 is an example of a class diagram, which partially specifies a university system. It captures the people hierarchy within the university and their relationship to the university courses. Classes are represented by rectangles; associations are represented by lines between the rectangles; cardinality constraints are marked on the association line ends; association classes are marked by a dashed line connecting a class rectangle with an association line; class hierarchy
constraints are marked by empty arrow heads; and association hierarchies are marked by a
dashed arrow labeled "subset" between association lines.

The semantics of class diagram assigns set extension\(^1\) to classes, and relations to
associations. For example, in Figure 1, the Academic class represents the set of academic people
in the university, and the association between FacultyMember and Course denotes a set of pairs
(links) of faculty members and courses in which the FacultyMember object plays the role of a
teacher. Attributes serve to identify all members of a class as having the same set of attributes
(and methods).

Constraints are used to restrict the otherwise unrestricted extensions of the components of
a class diagram. Class and association constraints restrict the set and relation extensions of
classes and associations, respectively. Attribute constraints restrict attribute values in terms of
types and cardinality. Association cardinality constraints are written at the association line end,
and specify the number of objects of one class that can be associated with one object from the
other class. For example, according to Figure 1, a FacultyMember can teach up to three Courses
(as indicated by the 0..3 cardinality constraint). Association classes restrict their objects to be
identified by pairs of the connected association. In Figure 1, every Enrollment object is identified
by a Course-Student object pair (no two enrollments are identified by the same pair).

Class
hierarchy
constraints
specify subset relation between the extensions of classes. In
Figure 1, the extensions of the FacultyMember and the Graduate classes are subsets of the
Academic extension. Class hierarchy constraints can be grouped into a Generalization Set, as
shown in Figure 1. For example, GraduateCourse, UnderGraduateCourse and Course form a
Generalization Set. In that case, more constraints can be defined on the group. There are two

\(^1\) The term set extension refers to the set of objects (class instances) that instantiate a class, and to the set of links
(association instances) that instantiate an association.
orthogonal planes for defining such constraints: (1) disjointness and (2) completeness. The constraints \{overlapping, complete\} on the generalization set of Course indicate that a course may be simultaneously a graduate and an undergraduate course \{overlapping\}, and that every course is either graduate or undergraduate \{complete\}.

![A partial class diagram of a university system](image)

**Figure 1. A partial class diagram of a university system**

### 2.1. Applications of UML Class Diagrams

UML models are now being used for a wide range of applications, including traditional software specification, model transformations, ontology design, and meta-modeling.

**System specification and model transformation:**

A class diagram is the backbone of any software specification since it captures the system structure throughout the lifecycle development. Moreover, in reverse engineering, the main output artifact is a class diagram that is a mirror of the code. The system specification is also used for transformation purposes such as generating code skeleton [64] and database schemata [13].

**Ontology design:**
Another emerging use of class diagrams is the design of ontologies [53, 56]. Currently, there is no standard for ontology representation languages. There are several OWL based dialects [8827] there is a WSML effort [26, 59], there are plain RDF written ontologies, there are general KIF based general ontologies like SUMO [79], and there are plenty industry initiated concrete ontologies (e.g., OAGIS [69] and Rosettanet [73]). One thing is agreed for all ontologies: Symbolic ontology specification is not human-understandable. Therefore, there is a need for a visual language for ontology specification and manipulation.

The UML class diagram model, being the standard for system specification that is widely adopted in the industry and supported by most CASE tools, is a natural candidate. The integration of the two formalisms is usually done using the transformation approach, i.e., defining rules for converting class diagrams to an ontology language specification [24, 25, 37, 40, 83].

**Meta-modeling:**

The class diagram model is a major meta-modeling formalism. A meta-model is an explicit model of the constructs and rules (abstract syntax) needed to build specific models within a domain of interest. A meta-model can be viewed from three different perspectives: (1) as a set of building blocks and rules used to build models; (2) as a model of a domain of interest, and (3) as an instance of another model [66]. Meta-modeling is the process of determining the requirements imposed on the modeling language or the modeling process [86]. The usage of meta-modeling has increased during the last decade. It is used for building meta-CASE tools for supporting multiple specification languages and composing new domain specific languages [6533]. It is also used as a means for specifying transformations among models, as in [2]. In addition, meta-models are used for data integration. The usage of UML class diagrams for meta-modeling
purposes have been demonstrated by Karsai et al., who use a subset of the UML class diagram and OCL for describing a meta-model [55]. The Eclipse environment for system development provides several plug-ins, e.g., the Generic Eclipse Modeling System (GEMS), that use class diagrams as a means for defining meta-models [31].

2.2. Reasoning Needs in UML Class Diagram

Class diagrams are models written by people, and therefore, usually suffer from modeling problems like inconsistency, redundancy, and abstraction errors. Inexperienced designers tend to create erroneous models, but even experienced ones cannot anticipate the implication of a change on an overall model [77]. Indeed, Lange et al. showed that model defects often remain undetected, even if practitioners check the model attentively [58].

These problems are empowered when a model originates from different resources, as frequently happens when Web services are integrated. Combined sources are usually overlapping, and the integration yields redundant inconsistent models [1, 15, 18, 44,46]. It is a common belief that such problems can best be solved at the level of models [50].

Thus, the need to provide coherent models is appealing. In particular, in order to avoid inconsistencies, redundancies, and ambiguities, it is essential to have tools that can support validation of the models. Furthermore, models can be improved, based on given design criteria. For ontology engineering purposes, it is of utmost importance to keep the model accurate, consistent, and unambiguous, as it usually serves multiple knowledge bases. The same holds for meta-models as they underlie the modeling of concrete systems. In order to achieve the goal of improving a model quality a diversity of reasoning capabilities is required.

Reasoning with formalism requires the existence of formal semantics that assigns an exact meaning to each expression of the formalism. Indeed, there is a wide research effort whose goal
is to provide formal semantics to class diagrams (and to UML as a whole). One way to define meaning is to directly assign a denotation to every language construct. This is termed declarative direct semantics. An alternative way to define language semantics is to use a different formally defined language that already has a formal semantics, as a "mediator". That is, instead of defining the meaning of elements directly as described above, they are translated into expressions of the intermediate language. This approach is termed indirect semantics.

The semantics of UML class diagrams has been defined using both approaches. The direct semantics of class diagrams assigns set extensions to classes and associations. It has been formally defined by several studies such as [72, 78]. There are several indirect semantics approaches for class diagrams in the literature. Full first order logic is the most popular intermediate language for class diagrams without OCL constraints [12, 54], or with OCL constraints [10, 70]. Some other intermediate languages are Z and Object-Z [22, 34, 35, 76], Algebraic specifications in the form of Abstract Data Types [3, 4], and Hierarchical Predicate Transition Nets [49].

In view of the wide spread usage of UML class diagrams and the difficulties of producing high quality models, it is essential to equip UML case tools with reasoning capabilities. The additional power can help in detecting design problems, identifying the reasons for these errors, suggesting possible solutions, and providing advice for design improvements [12, 52, 57, 85].

3. Reasoning Problems in Class Diagrams

Reasoning on UML models in general and on class diagrams, in particular, gains much attention, recently. Reasoning is necessary for improving the design quality and for supporting application construction needs. Design quality refers to (1) erroneous models that impose inconsistent constraints, (2) redundant models that can be simplified, and (3) models that can be improved
according to some design criteria [12, 46]. Reasoning helps in detecting erroneous models, finding the source of errors and possibly suggesting repairs. It is used for revealing redundant situations, and for testing whether design criteria are met.

Questions about class diagram quality deal with inconsistency, finite satisfiability, redundancy, and design improvement. Inconsistency arises when the constraints imposed by a class diagram are contradictory. Finite satisfiability is caused by cardinality constraints that can be satisfied by either empty or infinite class extensions (i.e., instantiations). Redundancy appears when constraints seem to allow values or links that cannot be realized (are inconsistent). Quality improvement deals with changing the models following various criteria such as design patterns or reuse enhancements.

Inconsistency and lack of finite satisfiability, are considered erroneous design. The first, because an inconsistent class diagram does not have a non-empty extension, and the latter, because there is no finite and non-empty extension [16]. For example, constraints that imply simultaneous disjointness and non-empty intersection of classes cause inconsistency (contradiction). Both questions may be asked at the diagram or at the class level. The consistency question at the whole diagram level is: "Is there an instance where all class extensions are non-empty?" Consistency at the class level is "For every class in a diagram, is there and instance in which the class extension is not empty?" Similarly, the whole diagram can be checked for finite satisfiability (i.e., having an instance where all class extensions are finite and non-empty), or whether a single class is finitely satisfiable. Most reasoning efforts have been devoted to the identification, detection and repair of inconsistency.

Redundancy is considered as inaccurate specification. For example, a wide cardinality constraint range, which cannot be realized without causing inconsistency, is a source of
redundancy. Equivalent classes (having the same extension) are another source of redundancy. Redundancy detection can help in simplification tasks, since it enables to tighten cardinality constraint ranges and removing redundant classes and associations.

*Design improvement* within a class diagram refers to the ability to represent the specification in terms of the appropriate abstraction for handling reuse and maintenance.

In the following we define and demonstrate the aforementioned problems. The various examples were kept small in size for the sake of clarity. These problems can clearly be indicated and resolved. Yet, when dealing with real system models (i.e., complex models), the identification and the solutions of such problems become a heavy-duty task.

### 3.1 Inconsistency (Unsatisfiability) in Class Diagrams

Inconsistency means class emptiness, that is, a class cannot be instantiated since the constraints imposed on its instances cannot be satisfied. The extreme case involves all classes in the diagram. Indeed in [12], Berardi et al. distinguish two cases:

**Consistency of a class diagram** – A class diagram is *consistent* (*satisfiable*) if it has an instantiation with at-least one non-empty class extension. Otherwise, it is *inconsistent* (*unsatisfiable*).

**Class consistency** – A class is *consistent* if there is an instantiation in which the class extension is non-empty. Otherwise, it is *inconsistent*.

We claim that class diagram consistency is of less importance, since every class diagram that has an unconstrained class is consistent.
Figure 2 presents an example of such a diagram. The Seminar class is unconstrained and can be freely instantiated.

![Figure 2. A consistent class diagram, due to an unconstrained class](image)

In fact, every realistic class diagram that we have checked proved to have such a class. Consequently, we suggest two additional notions termed *all class consistency* (satisfiability) and *full consistency* (satisfiability):

**All class consistency of a class diagram** – A class diagram is *all class consistent* (satisfiable) if every class is consistent.

**Full consistency of a class diagram** – A class diagram is *fully consistent* if it has an instance in which all class extensions are non-empty.

Clearly, *full consistency* implies *all class consistency*, which implies consistency. In the opposite direction, *Consistency of a class diagram* does not imply *all class consistency*. However, *all class consistency* implies *full consistency*. Intuitively, this claim is proved by the following argument: For every class, there is an instance in which the class extension is not empty. It can be shown that all these (disjoint) instances, can be combined into a single instance of the class diagram, in which all class extensions are non-empty. The argument holds due to the special character of UML class diagram constraints, which are invariant under disjoint instance combination [61, 62, 63].

Inconsistency is caused by constraint contradiction. UML class diagrams include seven kinds of constraints as described in Section 2. Kaneiwa and Satoh identify some factors for class
inconsistency [54]. We extend their work and characterize contradictory interactions between constraints.

1. **Inconsistency due to generalization set constraints:**

   Such inconsistency occurs when, for example, a generalization hierarchy has a disjoint constraint, where the disjoint subclasses have a common subclass. Figure 3(a) presents a case in which the disjointness constraint enforces the *Elective* class to be empty. In Figure 3(b), the interaction of the disjointness and the completeness constraints forces the class *B* to be empty. Any instance of *E* must be an instance of one of the subclasses *C* or *D*, but the classes *B*, *C* and *D* satisfy the *disjointness* constraint.

![Figure 3. Inconsistency due to generalization-set constraints](image)

2. **Inconsistency due to the interaction between class hierarchy constraints and cardinality constraints:**

   Such inconsistency may occur when contradictory cardinality constraints exist among associations that are related by a subset constraint. In Figure 4 the cardinality constraint of class *Member* within the *LargeComposition* is 5. Yet, as this association is defined as a subset of the *Composition* association which has cardinality constraint 1..3, there is a contradiction between these constraints, implying a necessarily empty extension for class *Large*.
3. Inconsistency due to the interaction between association constraints and cardinality constraints:

XOR constrains between associations contradict a non-zero minimum cardinality constraint since the XOR implies that only one of the associations is realized in every instantiation. Figure 4 presents such a situation.

3.2 Finite Satisfiabilty in Class Diagrams

Finite satisfiability means that a class has a finite non-empty extension. Lack of finite satisfiability is certainly erroneous since it means that the class cannot be instantiated. An example of a finite satisfiability problem can be derived from Figure 1 in Section 2. In this example, the cardinally and the class hierarchy constraints between classes Graduate and Academic enforce the Graduate class to be either empty or infinite.

Finite satisfiability is an essential property for all classes, since a necessarily empty or infinite class extension means that a class is either empty, and hence redundant, or its instantiation causes infinite loops. Similarly to the inconsistency case, finite satisfiability can
also refer to a single class or to the whole class diagram. However, while class diagram consistency requires the consistency of at least one class, finite satisfiability must apply to all classes.

In analogy with the terminology for inconsistency, we introduce three terms: *Class finiteness*, *all class finiteness*, and *full finiteness*.

**Class finite satisfiability** – A class is *finitely satisfiable* if there is a *finite instance* of the class diagram, in which the class extension is non-empty. A class diagram instance is finite if all class extensions are finite.

**All class finite satisfiability of a class diagram** – A class diagram is *all class finitely satisfiable* if for every class there is a finite instance in which the class extension is non-empty. Lenzerini and Nobili (1990) used the notion *strong satisfiability* for this term.

**Full finite satisfiability of a class diagram** – A class diagram is *fully finitely satisfiable* if it has a finite instance in which all class extensions are non-empty.

Clearly, full finite satisfiability implies all class finite satisfiability. The inverse is also true, i.e., all class finite satisfiability implies full finite satisfiability. The proof is similar to the consistency case. Note that in the consistency case no attention is paid to finiteness of the class diagram instances.

Finite satisfiability problems are caused by cycles of conflicting cardinality constraints. The cycles might include class hierarchy constraints, generalization set constraints, and association class constraints. We analyze six kinds of conflicting constraints that prevent finite satisfiability, and provide examples for each kind. In all cases the diagrams are fully consistent but lack finite satisfiability. That is, full consistency can be achieved only by infinite class extensions.
1. **Lack of finite satisfiability due to cardinality constraint conflict:**

In Figure 6, each course should have a single successor and at least two predecessors. Therefore, if the number of courses in a diagram instance is C, and the number of Dependency links is D, then D must satisfy \( D = C \times 1 \) and \( D \geq C \times 2 \), implying the inequality: \( C \geq C \times 2 \), that can be satisfied only by empty or infinite extensions for class \( \text{Course} \). Therefore, although the class \( \text{Course} \) is consistent it is not finitely satisfiable.

![Figure 6. Unsatisfiability due to cardinality constraint conflict](image)

2. **Lack of finite satisfiability due to class hierarchy and cardinality constraint conflict:**

Figure 7 presents a cardinality constraint cycle that involves a compound class, \( \text{Graduate} \), whose instances must be related to \( \text{Academic} \) instances. Therefore, using similar considerations as in the previous case, the number of student-advisor links in every diagram instance must be both, \( G \times 1 \) and \( A \times 2 \), assuming that \( G \) and \( A \) are the number of graduates and academics, respectively. Therefore, the extensions of \( \text{Graduate} \) and \( \text{Academic} \) must satisfy \( G = A \times 2 \), while the \( \text{Graduate} \) extension is a subset of the \( \text{Academic} \) extension. This constraint can, again, be satisfied only by empty or infinite extensions. Note that the diagram is fully consistent as every class has a non-empty extension.

![Figure 7. Unsatisfiability due to class hierarchy constraint conflict](image)
3. Lack of finite satisfiability due to interaction between an association class constraint and cardinality constraints:

Figure 8 presents a case where an association class, *Contract*, is constrained by contradictory cardinality constraints. In every legal instance, the number *C* of contracts is as twice the number of employees, *E*, since every employee is linked with two departments. Yet, the number of contracts is equal to the number of employees, as dictated by the *Manager* association. That is, *C* = *E*×2, and *E* = *C*, implying *E* = *E*×2 which can be achieved only by either empty or infinite extensions for all three classes.

![Figure 8. Unstatifiability due to association class and cardinality constraints conflicts](image)

4. Lack of finite satisfiability due to cardinality constraints conflict, that is caused by class and association hierarchy:

In Figure 9, classes *Member* and *Large* are related via associations *Management* and *LargeComposition*. The cardinality constraints in these associations can be finitely satisfied. However, the *LargeComposition* association is constrained to be a subset of the *Composition* association between *Team* and *Member*, implying that the cardinality constraint of *Member* in *LargeComposition* is actually tightened into 2..3.

This restricted cardinality range causes a finite satisfiability problem. For the *Management* association, if *M*, *L*, *Man* are the number of instances of *Member*, *Large* and *Management*, respectively, then *Man* = *L*×4, *Man* = *M*×1 imply *M* = *L*×4. For the *LargeComposition* association, if *LC* is the number of its links, then *L*×2 ≤ *LC* ≤ *L*×3, *LC* = *M*×1 imply *L*×2 ≤ *M* ≤
Replacing $M$ by $L^*4$ yields the inequality $L^*2 \leq L^*4 \leq L^*3$, that can be satisfied only if the $Large$, and thereby the $Member$, extensions are either empty or infinite.

5. **Lack of finite satisfiability due to asymmetry of aggregation and composition:**

The asymmetric property of aggregation/composition requires that a part of an assembly cannot aggregate one of its aggregators. Consequently, a legal instance of a class diagram cannot include aggregation cycles. Figure 10 depicts a case, where an organization consists of at least one child organization. Due to the acyclic character of the aggregation association, this cardinality constraint can be satisfied by either an empty or an infinite chain of organizations.

6. **Lack of finite satisfiability due to a generalization set constraint:**

In Figure 11 the {disjoint, incomplete} constraints suggests that the $Academic$ extension properly includes the $FacultyMember$ and the $Graduate$ extensions. Yet, $Academic$ elements are mapped in a 1:1 manner to $Graduate$ elements, implying that the sets have the same size. The problem is that non-emptyness of class $FacultyMember$ implies infinity of classes $Academic$ and $Graduate$. The only solution for proper set inclusion with equal size is that the sets are either empty or infinite.
3.3 Redundancy

Redundancy means that the specification can be simplified without affecting its meaning. For example, classes or associations that have the same extension in all instances are redundant. Another example involves cardinality values, in cardinality constraints or in attribute descriptions, which cannot be realized. In the first case, one of the equivalent elements can be removed. In the latter case, the cardinality value range can be tightened. In the following we demonstrate these two cases of redundancy.

1. **Redundancy due to equivalent classes**

Class redundancy occurs mainly due to generalization. In Figure 12 the *Graduate* extension is a subset of *Academic* due to the class hierarchy constraint, and the sets have the same size, since *Academic* elements are mapped in a 1:1 manner to *Graduate* elements. If the sets are finite, they must be equal, and therefore the *Graduate* class is redundant and can be removed.

![Figure 12. Redundancy due to equivalent classes](image)

2. **Redundancy due to equivalent associations**

Association equivalence can occur due to generalization constraints that involve association classes. In Figure 13, *R* and *Q* are associations and that constrain the association classes, *P* and *Q*, respectively. The association classes are constrained by class hierarchy, as well as by a 1:1
cardinality constraint. As in the previous case, \( P \) and \( Q \) have equal extensions in all diagram instances. Therefore, one of the associations \( R \) and \( Q \), and one of the association classes \( P \) and \( S \), are redundant, and should be removed.

![Figure 13. Redundancy due to equivalent associations](image)

**3. Redundancy of cardinality constraint values:**

Figure 14 demonstrates an association hierarchy constraint between the \textit{LargeComposition} and the \textit{Composition} associations. The cardinality constraint \( 1..3 \) of the \textit{Member} class in \textit{Composition}, makes the cardinality constraint \( 1..* \) of the \textit{Member} class in \textit{LargeComposition} too loose, as the maximal cardinality is 3.

![Figure 14. Redundancy due to cardinality constraints](image)

**3.4 Design Improvement**

Improving the design quality might arise when implicit consequences are noticed. For example, assume that a class diagram specification implies that a certain class or an association must be a sub-class or a super-class of another class or association, respectively. Then explicit specification of these hierarchy relations might improve the diagram quality, or expose relations that should be observed by the developer. Figures 15 and 16 demonstrate transformations of class diagrams,
based on redundancy and repair of semantic errors detection. Similar problems of class diagram redundancy and implicit consequences are identified and discussed by [12, 34, 78].

1. **Design improvement due to class redundancy:**

Figure 12 demonstrates redundancy since classes *Graduate* and *Academic* are equivalent (they necessarily have the same extension). The redundancy can be eliminated, thereby improving the overall design, as shown in Figure 15.

![Figure 15. Design improvement due to class equivalence](image)

2. **Design improvement due to incomplete hierarchical structure:**

Complex structures within class diagrams can include semantic errors, as shown in Figure 16a. The problem is that association hierarchy can apply only to class pairs that admit class hierarchy constraints (the subset relation can exist only among same type pairs). Instead of rejecting the diagram, a “smart” CASE tool might adopt a credulous approach, by inferring a repair to the detected problem, as shown in Figure 16b.

![Figure 16. Design improvement due to incomplete hierarchical structure](image)
4 Methods for Reasoning about Class Diagrams

Reasoning methods about class diagrams support both general implication and handling the problems of consistency, finite satisfiability, redundancy and design improvement, which are described in the previous section. Solutions to these problems can be at three levels: problem detection, cause identification, and repair [46]. Problem detection means just notification that a problem exists. Cause identification means detecting the reason for the problem, and repairing amounts to suggesting a solution. Most reasoning approaches provide problem detection alone. Yet, if we wish to have CASE tools that approach the level of current Integrated Development Environment compilers we need to provide at least cause identification.

Class diagram reasoning methods can be classified into concrete reasoning methods that directly solve specific problems [7, 43, 54, 61, 63], and to translation-based methods that support reasoning by mapping UML models into a formal reasoning framework [12]. Concrete methods tend to apply to error detection and revealing redundancy, while translation based methods deal with general query answering for a variety of modeling needs.

In the translation-based methods, a UML class diagram is translated into a formula or expression in some other language, and the translation is proved correct. The notion of correctness varies between studies. The formal notion requires a proof of equivalence, i.e., a proof that the translation preserves all and only the implications of the original class diagram. The main advantage of the translation-based approach is the uniform handling it provides to a variety of problems. Once the class diagram is translated, such methods rely on an already existing reasoner for question answering. Therefore, a single translation can serve for answering many questions. On the other hand, they cannot optimize solutions to problems.
Concrete methods present an opposite approach. Each method is usually targeted at solving a single problem, in an optimal way. Consequently, solutions are better dedicated to solve their problems, but many such solutions need to be designed. Yet, the concrete methods tend better to scale up to large problems, and it is easier to embed such algorithms within UML CASE tools. This option seems unreasonable for translation-based methods, as the embedding of a full reasoner tends to be quite heavy.

Before embarking on the description of existing methods, it is important to emphasize three essential features of reasoning methods: Soundness, completeness and complexity. Soundness means that the method infers correct results; completeness means that the method can derive all correct results, and complexity refers to its efficiency. It is well known that there is a tradeoff between the expressivity of a formalism and efficiency of reasoning on it. Greater expressivity leads to reduced efficiency. Reasoning on expressive formalisms is a hard problem, which cannot be solved efficiently. Therefore, insisting on complete reasoning necessarily leads to inefficient reasoning. Indeed, while soundness is unquestionable, completeness is a subject for debate: Should we (1) restrict the formalism, (2) abandon completeness, or (3) live with inefficient reasoning algorithms?

The reasoning methods described below demonstrate all of these options. The description logics based methods support complete, intractable reasoning on expressive class diagrams. Some concrete methods support complete tractable reasoning for restricted versions of UML class diagrams. Still other concrete methods provide efficient incomplete methods for reasoning over expressive class diagrams. Our survey separates the translation-based methods from the concrete ones. Among the first kind, we concentrate on the description logics based approach. The concrete methods are classified by the concrete addressed problems.
4.1 Translation-Based Methods for Reasoning about Class Diagrams

Using *Description Logics (DLs)* is, probably, the most prominent approach for reasoning about class diagrams and databases, by employing a general reasoner. It is motivated by the immediate similarity between the worlds described by both formalisms [14, 17, 19, 20, 21, 90].

Class diagrams describe a structured world of *classes* and *associations*, while DLs describe a similar world of *concepts* and *roles*. Beyond this analogy, each formalism has its own unique elements. Class diagrams have additional *desciptors* (*attributes* of classes and associations), and specific *constraints* that receive concrete visualizations. Description logics, being logic languages, are more uniform and have just few additional elements [5]. In fact, beyond concepts and roles, description logics have just *combinators* (*operators*) used for constructing *concept* and *role expressions*. Such expressions enable the introduction of an unbound number of implicit, nameless concepts and roles. These operators are the only difference between the various description logics. For example, referring to Figure 1, description logics can describe the following concepts:

- *Graduate-course or Undergraduate-course*\(^2\) – stands for all objects that are either graduate or undergraduate courses;
- *not (Faculty-member or Graduate)* – stands for all objects that are not academics;
- *Course and all Enrollment is (atleast 2 Graduate and Resident)* – stands for all courses having at least two enrolled graduate resident students;
- *Course and exists teacher.student is Resident* – stands for all courses whose teacher advises a resident graduate student;

\(^2\) For the sake of readability, we use an intuitive free syntax.
The common feature of the above four concepts is that they are nameless, and are implicitly defined using the concept forming operators **or, and, not, all, exists, atleast**\(^3\). This is different from UML class diagrams, where the only way to introduce classes is via an explicit new name, that is associated with the class attributes, associations and constraints.

Implicit concept/role construction opens the door for targeted concept/role reasoning. For example, it is possible to infer that:

- Objects that are *neither* faculty members nor graduates are necessarily not academics: \( \text{not } (\text{Faculty-member or Graduate}) \text{ is subsumed by not Academic}; \)

- Electives that have at least two student enrollments are not necessarily courses with at least one graduate student enrollment:

  \[
  \text{Elective and all Enrollment is (atleast 2 Student)} \text{ is not subsumed by} \\
  \text{Course and all Enrollment is (atleast 1 Graduate)};
  \]

Indeed, description logics have developed concrete sound and complete inference algorithms, that are implemented in state of the art description logic reasoners like Racer [41] and FaCT [51]. Moreover, FaCT++ [84] supports reasoning with ontology languages like OWL [71]) and WSML-DL [26].

The ontological similarity between UML class diagrams and description logics, and the essential reasoning capabilities of description logic tools present DLs as a natural candidate for supporting reasoning about class diagrams. Berardi et al. encode class diagrams in description logics and prove that reasoning on UML class diagrams is EXPTIME-hard [12]. They also show that under minor restrictions, UML class diagrams can be encoded in the description logic \(\text{ALCQI}\) which is the most expressive description logic that is supported by DL reasoning tools. Berardi describes a tool that translates UML class diagrams into the DL reasoners FaCT and

---

\(^3\) This is similar to a relational DB table, obtained by a relational algebra operator from other tables.
Racer [11]. The ICOM tool supports DL based reasoning on Extended Entity Relationship and UML class diagrams [38].

The encoding of UML class diagrams into Racer suggests that UML CASE tools can provide reasoning capabilities by association with DL reasoners. The beauty of such encoding is that it can be used for answering a variety of questions about a class diagram. In fact, apart from finite satisfiability, all reasoning problems discussed in this paper can be answered using the DL translation. Moreover, using the DL encoding, all reasoning problems can be reduced to the emptiness problem.

Nevertheless, the description logic based reasoning does not support inconsistency cause identification and certainly not repairing. In order to localize the reason for inconsistency, one has to query for the consistency of each class, separately. Moreover, reasoners would not suggest implicit consequences by themselves, unless they are explicitly asked for. Therefore, DL based reasoners for reasoning on UML class diagrams cannot function as smart compilers that can detect the cause for a problem and suggest solutions for repairing.

A different translation based approach for reasoning about class diagrams is reported in [3, 4]. In that study, UML class diagrams are translated into abstract data types, and the Larch prover [39] is used for reasoning about them. Similarly to the description logics based translation, the method enables reasoning on a variety of inconsistency problems. The problem seems to lie in scaling, as each class diagram element translates into multiple algebraic rules, and the prover cannot handle large sets of rules.

4.2 Concrete Methods for Reasoning about Emptiness (inconsistency) of Class Diagrams
Kaneiwa and Satoh study the problem of full consistency in a subset of UML class diagrams that include classes with typed attributes and cardinality constraints on the attributes, unconstrained associations and constrained generalization sets [54]. They identify three factors for inconsistency in such diagrams: (1) combination of generalization with disjointness; (2) attribute overwriting in multiple hierarchies; and (3) combination of completeness and disjointness constraints in generalization sets. Based on these factors, they provide tractable algorithms for deciding full consistency in the restricted class diagram model.

The algorithms operate on a first order logic encoding of the class diagram, which serves as an indirect-semantics definition. They analyze the structure of the given diagram and identify occurrences of the inconsistency factors. Consequently, this concrete method functions as inconsistency detection algorithm and cause identification algorithm as well.

This method is implemented as a debugging system for restricted class diagrams [75]. The debugging is based on an elaborated set of rules for contradiction detection. The rules enable identification of the part in the UML diagram that cause the inconsistency, and suggest a possible solution.

In comparison with the description logic based approach for deciding consistency, this method is stronger in the sense that it can decide full consistency in a tractable time, identify its cause, and suggest a solution. Yet, the method applies only to a restricted version of UML class diagrams. Description logics, on the other hand, account for any class diagram, but cannot account for full consistency using a single query. One can query for consistency of every class in the diagram, but not for the simultaneous consistency of all classes.

4.3 Concrete Methods for Reasoning about Finiteness of Class Diagrams
Reasoning on finiteness of entity relationship and class diagrams has attracted much attention. The problem was independently identified by Lenzerini and Nobili [61] and by Thalheim [8080, 81, 82], and referred to entity relationship diagrams. Later on the methods have been extended to various fragments of UML class diagrams. The problem is to detect, identify cause and suggest repair, to diagrams that are not fully finitely satisfiable.

There are two main approaches: the linear programming approach and the graph based approach. The first approach reduces the full finite satisfiability problem to the problem of finding a solution to a system of linear inequalities. The second approach detects infinity causing cycles in the diagram, and possibly suggest repair transformations. All methods apply only to fragments of UML class diagrams. Detection of infinity in unrestricted UML class diagrams is still an open issue.

4.3.1 Linear Programming Based Full Finite Satisfiability Detection:

The work of Lenzerini and Nobily (1990):

The fundamental method of Lenzerini and Nobily [61] is defined for an entity relationship diagram that includes Entity types (Classes), n-ary Relationship types (Associations), and Cardinality Constraints. The method consists of a transformation of the cardinality constraints into a set of linear inequalities whose variables stand for the sizes (cardinalities) of the entity and relationship types in a possible instance. A relationship:

\[ C_2 \xrightarrow{min_1, max_1, r, min_2, max_2} C_1 \]

yields the four inequalities: \( r \geq min_1 \cdot C_1 \), \( r \leq max_1 \cdot C_1 \), \( r \geq min_2 \cdot C_2 \), \( r \leq max_2 \cdot C_2 \), where \( r, C_1, C_2 \) are variables that stand for the sizes of the respective entity or relationship types. In addition,

---

4 Lenzerini and Nobili (1990) use the membership semantics for cardinality constraints (consult Balaban and Shoval (2002) for semantics of cardinality constraints). For non-binary relationships, this is not the standard semantics of cardinality constraints, neither in the entity relation model nor in the class diagram model.
for every entity or relationship type $T$, the inequality $T > \theta$ is inserted. The size of the inequality system is polynomial in the size of the diagram. The main result is that the entity relationship diagram is fully finitely satisfiable if and only if the inequalities system has a solution. Since linear programming is solvable in polynomial time in the size of the problem encoding, full finite satisfiability for this fragment of class diagrams can be decided in polynomial time.

**The work of Calvanese and Lenzerini (1994):**

Calvanese and Lenzerini extend the inequalities based method of Lenzerini and Nobili [63] to apply to schemata with ISA (class hierarchy) constraints [17]. The expansion is based on the assumption that class extensions may overlap. They provide a two stage algorithm in which the full finite satisfiability problem of a class diagram with ISA constraints is reduced into the full finite satisfiability problem of a class diagram having no class hierarchy constraints. Then, similarly to Lenzerini and Nobili [63], the full finite satisfiability of the new class diagram is checked by testing whether a derived linear inequalities system has a solution.

The linear inequalities system derived in Calvanese and Lenzerini [17] is different from the one derived by Lenzerini and Nobili [63]. It is quite complex, and might introduce, in the worst case, an exponential number, in terms of the input diagram size, of new classes and associations. This method was simplified in Cadoli et al. by restricting class overlapping to class hierarchy alone [16]. The latter simplification reduces the overall numbers of classes and association, but still introduces, in the worst case, an exponential number of new classes and associations.

Calvanese and Lenzerini show that class hierarchy constraints can be used also for answering *implication queries* [17]. That is, answering whether a constraint is implied by a class diagram. The idea is to add an ISA constraint in which the desired implication is embedded. For
example, to answer questions like "Is $m$ the minimum (maximum) cardinality on the role $R$ of a class $C$ in an association $A$", they suggest to add a subclass $C'$ to $C$, with a maximum $m-1$ (minimum $m+1$) for its role $R$ in $A$, and ask for finite satisfiability of the new class. The implication holds if and only if the new class $C'$ is not finitely satisfiable (because in that case, in every extension of $C$ the cardinality is as required).

**The work of Balaban and Marae (2006):**

Balaban and Marae extend the inequalities based method of Lenzerini and Nobili [63] to apply to UML class diagrams with (1) binary association; (2) class hierarchy (ISA) constraints; (3) generalization set constraints disjoint/overlapping and complete/incomplete; (4) n-ary association with the standard interpretation of cardinality constraints; (5) qualifier dependent cardinality constraint; (6) association classes; (7) association hierarchy constraints [7, 62, 63]. The extension is based on a preprocessing reduction of full finite satisfiability of a given class diagram, to the full finite satisfiability of a restricted class diagrams handled by the Lenzerini and Nobili method [63].

The generalization set constraints present the hardest requirement. Their occurrences require the addition of new inequalities, and restrict the method scope to class hierarchies whose undirected graph structure is acyclic (termed below *acyclic class hierarchies*). For example, for the class diagram in Figure 1, the method presented in [62, 63] does not apply to the *Course* cyclic class hierarchy, but fully applies to the rest of the diagram.

The advantage of this method over the Calvanese and Lenzerini method [19] lies in its simplicity and efficiency. The method introduces only a linear number of new associations and inequalities (linear in the diagram size) and requires only a linear inequalities solver. Therefore it is simple to extend a UML CASE tool with this method [63]. On the other hand, this method
does not apply to cyclic class hierarchies (see above). Therefore, it seems reasonable to combine it with the Calvanese and Lenzerini method [19]. That is, use the first method for the major part of a class diagram, and apply the latter more expensive method to the cyclic class hierarchies in the diagram. This method integration relies on the assumption that cyclic class hierarchies do not occur frequently in class diagrams.

The three linear inequalities based methods described above act only as detectors for full finite satisfiability. Boufares and Bennaceur [15] suggest using the Fourier-Motzkin elimination method [89] for solving the obtained system of linear inequalities. They show how this method can help in identifying the source of infinity, when the inequalities system is unsolvable. The idea is that backtracking the solution process reveals the conflicting cardinality constraints.

4.3.2 Graph-Theoretic Full Finite Satisfiability Detection, Cause Identification and Repair:

The work of Lenzerini and Nobili (1990):

In addition to the linear inequalities method, Lenzerini and Nobili also present a method for identification of causes for lack of finite satisfiability [63]. For that purpose they convert the diagram into a directed graph whose nodes stand for classes and relationships. The edges are assigned weights that are derived from the cardinality constraints. They prove a connection between finite satisfiability to the weights on graph cycles. Cycles that imply infinity are termed critical cycles. Moreover, each critical cycle singles out a non-finitely satisfiable set of cardinality constraint. Lenzerini and Nobili were the first to suggest a method for cause identification of full finite satisfiability in restricted entity relationship diagrams. Their solution is not constructive, as they do not provide a method for computing critical cycles. A first step towards finding critical cycles appears in [82]. Hartman presents a different method for identifying the cause for infinity [43] (see below).
**The work of Dullea and Song (1998):**

Dullea et al. characterize infinity causing structures (termed *structural invalidity*) of recursive binary and ternary relationship types in entity relationship diagrams [29, 30]. The analysis suggests a set of structure based decision rules for identifying structural invalidity in entity relationship diagrams.

**The work of Hartman (1995; 2001a; 2001b):**

Hartman handle the problems of full finite satisfiability from all three aspects of detection, cause identification and repair [43, 46, 47]. He suggests a polynomial time graph-theoretical method for the detection of full finite satisfiability in entity relationship diagrams (same model as in Lenzerini and Nobili). He defines a similar notion of critical cycles, and uses it for detecting infinity problems. In addition, for graphs without critical cycles, the method can derive a minimal finite instance. More details on this work appear in [44, 45].

Hartman suggests critical cycles based methods for cause identification and repairing for lack of finite satisfiability problems [46]. The paper suggests four heuristic strategies for repairing infinity problems. One method is based on finding minimal inconsistent constraint sets that exist in every critical cycle, and suggests how to repair them. Another method is based on the notion of feedback arc set which is a set of arcs that intersects all critical cycles in the graph. The paper suggests a cardinality constraint repair plan, based on an optimization method for finding a minimal feedback arc set.

Hartmann extends the former work for reasoning about a set of cardinality constraints, key constraints, soft constraints and functional dependencies [47, 48]. He presents a list of seven implication rules for deriving new cardinality constraints from given ones.
5. Concluding Remarks

Reasoning on UML models in general and on class diagrams, in particular, gains much attention, recently. In this paper we have classified the major problems that require reasoning on UML class diagrams, and surveyed the existing approaches and results. We claim that reasoning is essential for supporting advanced IDE and CASE tools in all application areas of UML class diagrams. The emerging model driven development approach requires reliable models that are equipped with powerful reasoning capabilities.

We have identified four categories of reasoning problems: inconsistency, finite satisfiability, redundancy, and design improvement. The reasoning techniques survey distinguishes between transformation-based approaches and concrete methods. In particular, we surveyed reasoning methods that concern emptiness and finite satisfiability problems in class diagrams.

Transformation-based approaches tend to be inclusive, powerful, and solve a variety of problems. Yet, they fall short of handling full consistency, finite satisfiability, and cause identification and repair for inconsistency. In particular, weaving such reasoners into an IDE or a CASE tool is quite heavy. Concrete reasoning methods are, usually lighter, scalable (complexity depending), and easily weaved into existing tools. Yet, they do not apply to unrestricted class diagrams. In general, cause identification and repair did not receive sufficient attention.

We envision that the next generation of reasoning tools will embark on method integration. Tools would employ a mixture of reasoning methods, applying the simpler methods where possible, and resorting to heavy translation-based reasoning when other methods fail. Embedding reasoning capabilities within these tools requires scalability; for that purpose incremental reasoning needs to be further explored.
References

64. Martin, R.C., UML for Java™ Programmers (Prentice Hall, 2003).