

# Management of Correctness Problems in UML Class Diagrams – Towards a Pattern-based Approach

Mira Balaban<sup>1</sup>, Azzam Maraee<sup>1</sup>, Arnon Sturm<sup>2</sup>

<sup>1</sup>Department of Computer Science, <sup>2</sup>Department of Information Systems Engineering,

Ben-Gurion University of the Negev, Beer Sheva 84105, Israel

[mira\\_mari@cs.bgu.ac.il](mailto:mira_mari@cs.bgu.ac.il), [sturm@bgu.ac.il](mailto:sturm@bgu.ac.il)

## Abstract

UML is now widely accepted as the standard modeling language for software construction. The *Class Diagram* is its core view, having well formed semantics and providing the backbone for any modeling effort. Class diagrams are widely used for purposes such as software specification, database and ontology engineering, meta-modeling, and model transformation. The central role played by class diagrams emphasizes the need for strengthening UML modeling tools with features such as recognition of erroneous models and the detection of errors' sources.

Correctness of UML class diagrams refers to the capability of a diagram to denote a finite but not empty *reality*. This is a natural, unquestionable requirement. Nevertheless, incorrect diagrams are often designed, due to interaction of contradicting constraints and limitations of current tools. In this paper we clarify the notion of class diagram correctness, discuss various approaches for detecting correctness problems, and propose a pattern-based approach for identifying situations in which correctness problems occur and for providing explanations and repair advices.

**Keywords:** UML class diagrams, finite satisfiability, reasoning, detection, identification, correctness, patterns, model driven engineering

## 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 (OMG-UML, 2006). It consists of several diagrammatic languages, each describing a different view of object-oriented software; a system model consists of a collection of such diagrams. The most important view of UML is the

static/structural specification which describes a structural abstraction of the real world. This view is expressed by class diagrams, which consist of *classes* and their *descriptors*, *associations* among them, and *constraints* imposed on both classes and associations. Among the nine visual UML models, class diagrams appear to be the most clear, intuitive and well defined.

Dobing and Parsons (2006) found that the Class Diagram view is the most frequently used (73%) as they examine the usage of UML. It was found useful in clarifying technical understanding, and for maintaining software documentation. The major usage of UML class diagrams is to specify, visualize, and document systems' static view. They also serve as a basis for generating implementation artifacts such as code skeleton (Martin, 2006) and database schemata (Blaha et al., 1994), as a means for knowledge representation such as specifying ontologies (Cranefield, 2001; Falkovych et al., 2003; Gasevic et al. 2004; Kabilan& Johannesson, 2004; Kogut, 2002; Timm & Gannod, 2005), and for defining meta-models of other programming, modeling, and specification languages.

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 (Sunye et al., 2001). Indeed, Lange et al. (2006) show that model defects often remain undetected; even if practitioners check the model attentively. These problems are empowered when a model originates from different resources. Combined sources are usually overlapping, and the integration yields redundant inconsistent models (Ackermann & Turowski, 2006; Huzar et al., 2004). It is a common belief that such problems can best be solved at the level of models (Jackson & Rinard, 2004).

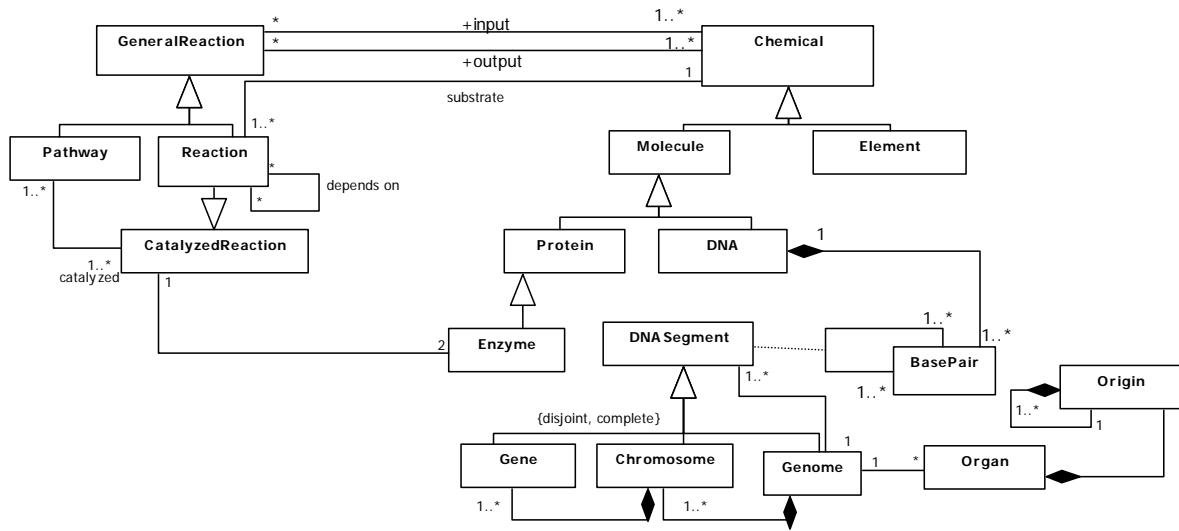
In view of the wide spread usage of UML class diagrams and the difficulties of producing high quality models, it is essential to equip UML CASE tools with reasoning capabilities for identifying problems within models (Berardi et al., 2005; Cadoli et al., 2004; Hartman, 2001; Jackson & Rinard, 2004). The additional power can help in detecting design problems, identifying the reasons for these errors, suggesting possible solutions, and providing advice for design improvements (Unhelkar, 2005). The quality of models is especially important for the emerging *Model Driven Engineering (MDE)* approach, in which software is developed by repeated transformations of models (Stahl et al., 2006).

The major correctness features of class diagrams involve *consistency* and *finite satisfiability*. They guarantee the natural, unquestionable requirement, that a diagram can denote a finite but non empty reality: Consistency accounts for *non-emptiness*, and finite satisfiability accounts for *finiteness*. Emptiness is caused by contradicting constraints, such as designing a subclass of two necessarily disjoint classes. Non-finiteness is caused by interaction among *multiplicity (cardinality) constraints*, which restrict the number of interactions between objects of related classes. Contradicting multiplicity constraints, or a contradiction between multiplicity constraints to other constraints, like generalization and association class constraints, impose class multiplicity requirements that can be satisfied only by empty or infinite classes. For example, the class diagram in Figure 1 is not finitely satisfiable, since in every legal instance of this diagram the sets denoted by the classes *CatalyzedReaction*, *Enzyme*, *Protein*, *Molecule*, *Chemical*, and *GeneralReaction*, are either all empty or all infinite.

In this paper we define the *consistency* and *finite satisfiability* problems in class diagrams, describe current approaches for detection of these problems and identification of their cause, and suggest a *pattern-based* approach for creating explanations and repair advices. Section 2 presents the class diagram model, its semantics and correctness problems. Section 3 surveys existing methods for management the correctness of class diagrams. Section 4 presents a pattern-based approach for detecting correctness problems within class diagrams for providing explanations and repair advices. Section 5 summaries the paper and draws the line for future research.

## 2. UML Class Diagrams

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) *multiplicity (cardinality) constraints on associations*, with or without *qualifiers*; (2) *class hierarchy constraints*; (3) *generalization set constraints*; (4) *association class constraints*; (5) *inter-association constraints*; (6) *aggregation constraints*; and (7) *multiplicity constraints on attributes*. The syntax and informal semantics are described in OMG-UML (2006) and Rumbaugh et al. (2005).



**Figure 1. A class diagram describing a partial ontology in the molecular biology domain**

Figure 1 is an example of a class diagram, which partially specifies ontology in the molecular biology domain. It demonstrates the above constraints, apart from multiplicity constraints on attributes (no attributes in the diagram). The constraints are inter-woven in a complex way: Class *DNA Segment* is involved in an association class constraint, class hierarchy, generalization set constraint and a multiplicity constraint. Its subclasses are involved also in aggregation constraints.

The *semantics* of a class diagram is given by its *legal instances*. An *instance* of a class diagram is an assignment of: (1) set extensions<sup>1</sup> to classes; (2) relations among these classes to associations; and (3) value mappings to attributes. A *legal instance* is an instance that satisfies all constraints in the diagram. For example, in Figure 1, the *Chemical* class represents the set of chemicals within a cell, and the association between *Chemical* and *Reaction* denotes a relation (a set of links) between the *Chemical* extension and *Reaction* extension. In a legal instance of Figure 1, every *Element* object must also be a *Chemical* object, and be related to at least one *Reaction* object.

*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

---

<sup>1</sup> 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.

multiplicity. *Association multiplicity constraints* specify the number of objects of one class that can be associated with one object from the other class.

*Hierarchy constraints* specify subset relations between the extensions of classes or associations. In Figure 1, the classes *DNASegment*, *Gene*, *Chromosome* and the *Genome* form a *Generalization Set (GS)*, with *super-class DNASegment*, and *subclasses Gene, Chromosome and Genome* are presented. It states that the subclass extensions are subsets of the super-class extension. The *GS* is constrained by the *GS constraint {complete, disjoint}*. *GS* constraints describe (1) *disjointness* (or lack of – *overlap*) among the subclasses, and (2) *completeness* (or lack of – *incomplete*) of covering the super-class. The *{complete, disjoint}* constraint on the above *GS* indicates that in every legal instance, the extensions of the *Gene, Chromosome, and Genome* classes are disjoint, and their union covers the extension of the *DNASegment* class.

An association class constraint specifies a 1:1 mapping between the extensions of the association class and the related association. In Figure 1, such a mapping must exist between the extensions of the *DNASegment* class and the unary *BasePair* association. Inter association constraints can enforce hierarchy or disjointness among associations. Aggregation constraints enforce whole-part constraints.

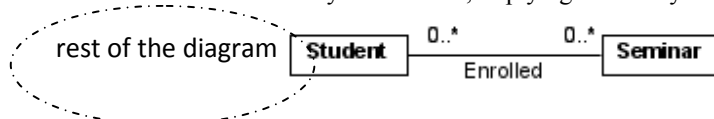
Correctness of class diagrams involves *consistency* and *finite-satisfiability*. A class is *consistent* if it has a non-empty extension in some legal instance; a *class diagram is consistent* if all of its classes are consistent<sup>2</sup>. A class diagram instance is finite if all class extensions are finite.

A class is *finitely-satisfiable* if it has a non-empty extension in a legal finite instance; a *class diagram is finitely satisfiable* if all of its classes are finitely satisfiable<sup>3</sup>. The class diagram in

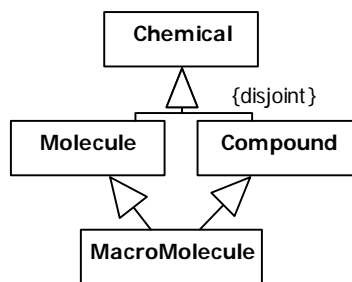
Figure 2 is inconsistent since the *{disjoint}* constraint on the generalization set *{Chemical, Molecule, Compound}* enforces the class *MacroMolecule* to be empty, implying that all of its subclasses are also empty.

---

<sup>2</sup> Berardi et al. (2005), introduce a weaker notion of consistency. They define a class diagram as consistent if it has a legal instance with at-least one non-empty class extension. But this definition misses the point of consistency, since it implies that every class diagram that has an unconstrained class is consistent. In fact, every realistic class diagram that we have checked proved to have such a class. In the following diagram, the *Seminar* class is unconstrained and can be freely instantiated, implying that every class diagram that includes it is consistent.



<sup>3</sup> Lenzerini and Nobili (1990) and Thalheim (1993) used the term strong satisfiability for this notion.



**Figure 2: A class diagram with inconsistency problem.**

It can be shown that if a class diagram is consistent, then there exists a legal instance in which all class extensions are non-empty, and if the class diagram is finitely satisfiable, then there is a legal finite instance in which all class extensions are non-empty (Lenzerini, M. & Nobili, 1990; Maraee, 2007; Maraee et al., 2008). Therefore, in order to check consistency it is sufficient to search for a single legal instance, in which all classes are non-empty, and in order to check finite satisfiability it is sufficient to show that this instance is also finite. Note that finite satisfiability requires consistency.

### **Complexity:**

Berardi et al. (2005) show that deciding consistency of UML class diagrams is EXPTIME-complete. The proof is obtained by providing consistency and finite satisfiability preserving reductions to/from hard *Description Logics (DLs)*: From *ALC*, which is the least expressive description logic, and into *ALCQI*, which is the most expressive description logic that is supported by tools (Haarslev & Möller, 2001; Horrocks, 1998; Tsarkov & Horrocks, 2006) (in fact, the reduction into *ALCQI* considers UML class diagrams under some minor restrictions). Artale et al. (2007) refine these results, by considering fragments of ER diagrams. As for finite satisfiability, it was shown that for *ALCQI*, it is EXPTIME-complete (Carsten et al., 2005). Since for *ALC*, finite satisfiability and consistency coincide (Schild, 1992), it follows that finite satisfiability of UML class diagrams (under minor restriction) is also EXPTIME-complete (Berardi et al., 2005; Carsten et al., 2005; Schild, 1992).

## **3. Survey of Correctness Handling Techniques for Class Diagrams**

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. Inconsistency is caused by contradictory constraints that cannot be simultaneously satisfied. Hartman (2001) specifies three levels of reasoning about these problems: *Problem Detection*, *Cause Identification*, and *Advice*<sup>4</sup>. Problem detection refers to notification that a problem exists. Cause identification means pointing to the problem source (like advanced IDE compilers), and advise amounts to suggesting a solution. Most reasoning approaches provide problem detection alone.

Consistency of class diagrams have been handled by translation to other reasoning frameworks. The most notable approach is the translation of UML class diagrams into a description logic and activation of a description logic reasoner for determining consistency (Berardi et al., 2005; Cadoli et al, 2004). Other approaches combine reasoning over class diagrams with OCL reasoning. We are not aware of direct techniques for reasoning about class diagram consistency.

Reasoning on finite satisfiability of entity relationship and class diagrams has attracted much attention. The problem was independently identified by Lenzerini and Nobili (1990) and by Thalheim (1993), and referred to entity relationship diagrams. Later on the methods have been extended to various fragments of UML class diagrams.

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

### 3.1 The Linear Inequalities Approach

The fundamental method of Lenzerini and Nobili (1990) and Thalheim (1993) is defined for an entity relationship diagram that includes entity types (classes), n-ary relationship types (associations), and multiplicity constraints<sup>5</sup>. The method consists of a transformation of the multiplicity constraints into a set of linear inequalities whose variables stand for the size of the

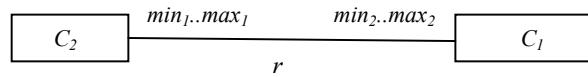
---

<sup>4</sup> Hartman uses the terminology *Problem Recognition*, *Cause Detection*, *Problem Repair*.

<sup>5</sup> 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.

entity and relationship types in a possible instance. The binary association in Figure 3 yields the following inequalities:

- $C_1 > 0; C_2 > 0; r \geq 0;$
- For  $min_1 \neq 0$ :  $r \geq min_1 \cdot C_1;$  for  $max_1 \neq \infty$ :  $r \leq max_1 \cdot C_1;$
- For  $min_2 \neq 0$ :  $r \geq min_2 \cdot C_2;$  for,  $max_2 \neq \infty$ :  $r \leq max_2 \cdot C_2;$



**Figure 3: A binary association**

The rationale behind these inequalities is that in order to satisfy these constraints there must be at least  $min_2 \cdot C_1$  and  $min_1 \cdot C_2$ , and at most  $max_2 \cdot C_1$  and  $max_1 \cdot C_2$  links in the relationship  $r$ , since every  $C_1$  instance is related to at least  $min_1$  and at most  $max_1$  instances of  $C_2$ , and vice versa for  $C_2$ . In addition, for every entity  $C$  or relationship type  $R$ , the inequalities  $c > 0$  and  $r \geq 0$  are inserted. The size of the inequality system is polynomial in the size of the diagram.

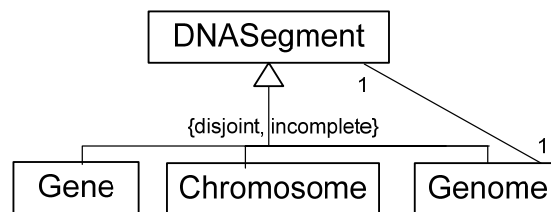
The main result is that the entity relationship diagram is finitely satisfiable if and only if the inequality system has a solution. Since linear programming is solvable in polynomial time in the size of the problem encoding, finite satisfiability for this fragment of class diagrams can be decided in polynomial time.

Calvanese and Lenzerini (1994) extend the inequalities based method to apply to diagrams with class hierarchy constraints. The expansion introduces a variable for every possible class intersection among subclasses of a super-class, and splits relationships accordingly. Therefore, the size of the resulting system of inequalities is exponential in the size of the class diagram.

Maraee and Balaban extend the method of Lenzerini and Nobili (1990) and Thalheim (1993) to apply to diagrams with class hierarchy constraints, *GS* constraints, qualifier constraints, and association class constraints (Maraee, 2007; Maraee & Balaban, 2007; Maraee & Balaban, 2008; Maraee et al., 2008). The extension is summarized in the *FiniteSat* efficient algorithm for deciding finite satisfiability in UML class diagrams. The scope of the algorithm is defined by the structure of the class hierarchy in a knowledge base, rather than by a fragment of the language.

**Example:** The application of *FiniteSat* to the class diagram in Figure 4, yields the inequality system below. We use the variables  $ds$  for *DNASegment*,  $gn$  for *Gene*,  $ch$  for *Chromosome*,  $gm$  for *Genome* and  $ds-gn$  for the *DNASegment-Genome* association.

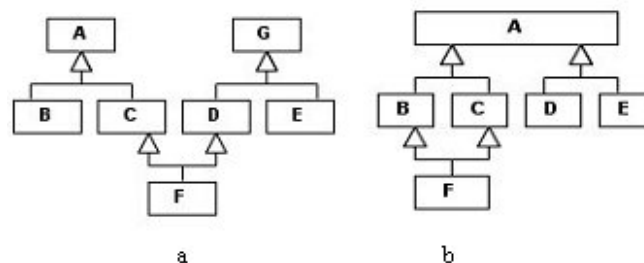
- **Multiplicity constraints:**  $ds-gn \geq 1 \cdot ds$ ,  $ds-gn \leq 1 \cdot ds$ ,  $ds-gn \geq 1 \cdot gn$ ,  $ds-gn \leq 1 \cdot gn$
- **Class hierarchy constraints:**  $ds \geq gn$ ,  $ds \geq ch$ ,  $ds \geq gm$
- **GS constraint:**  $ds > gn+ch+gm$
- **Non emptiness inequalities:**  $ds, dn, ch, dm, dg > 0$ ,  $ds-gn \geq 0$



**Figure 4. Non-finite satisfiability due to a generalization set constraint**

This system has no solution since the multiplicity inequalities imply  $ds = gn$ , while the GS constraint and the non emptiness inequalities require that  $ds > gn$ . Therefore, *FiniteSat* returns False.

Correctness of the *FiniteSat* depends on the structure of class hierarchies in the given class diagram. For that purpose, class hierarchy constraints are viewed as a graph (directed or not), whose nodes represent classes and its edges represent class hierarchy constraints, directed from super-classes to their subclasses. Three class hierarchy structures are analyzed: (1) *Tree class hierarchies*: A tree structure as in Figure 1; (2) *Acyclic class hierarchies*: The undirected graph is acyclic (a tree), as in Figure 5a; (3) *Cyclic class hierarchies*: The undirected graph is cyclic, as in Figure 5b, implying unrestricted multiple inheritance.

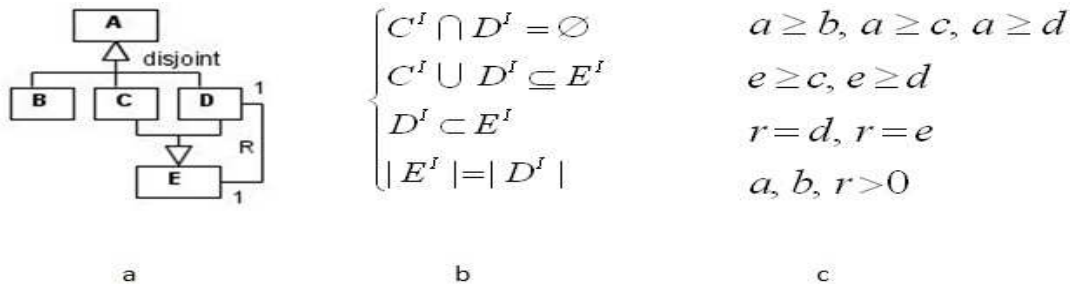


**Figure 5. Unconstrained Hierarchy Structures**

The correctness results for *FiniteSat* are as follows:

1. If the class hierarchy structure in a class diagram  $CD$  does not include cycles with a *disjoint* or *complete GS* constraint, then  $CD$  is finitely satisfiable if and only if the inequality system constructed by *FiniteSat* is solvable.
2. If the class hierarchy structure in a class diagram  $CD$  includes cycles with a *disjoint* or *complete GS* constraint, then  $CD$  is not finitely satisfiable if the inequality system constructed by *FiniteSat* is unsolvable.

**Example: [*FiniteSat* Limitation]** The class diagram in Figure 6a is not finitely satisfiable, since it implies the semantic inter-relations shown in Figure 6b. Yet, *FiniteSat* yields the solvable inequality system in Figure 6c. The reason for the failure of *FiniteSat* lies in the projection of the *disjoint* constraint from the  $A$  GS to the  $E$  GS, which is not recorded in the inequality system. Recently, *FiniteSat* was strengthened with propagation of *disjoint* and *complete GS* constraints (Maraee & Balaban, 2008).

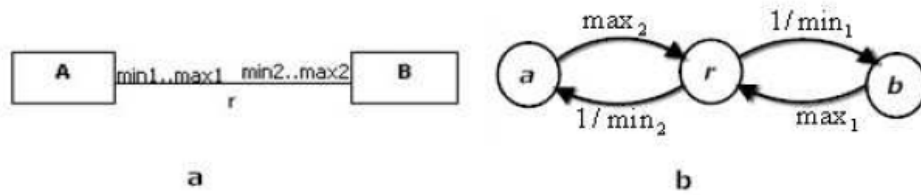


**Figure 6. A Finite Satisfiability Problem that is not Recognized by the FiniteSat Algorithm**

The construction of the inequalities by *FiniteSat*, and their number is  $O(n^2)$ , where  $n$  is the number of constraints in the class diagram. The size of the inequality system is  $O(n)$  if there are no association class hierarchies.

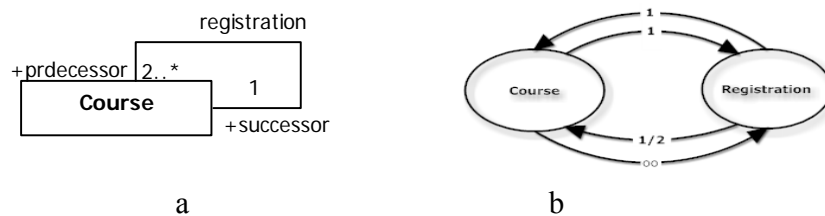
### 3.2 The Detection Graph Approach

A method for detection of the cause for non finite satisfiability is suggested in Hartmann (2001), Lenzerini & Nobili (1990) and Thalheim (1993). The method is based on construction of a directed graph, termed *the identification graph*, whose nodes stand for classes and associations, and its edges connect association nodes with their end class nodes. The edges are weighted by the multiplicity constraints, as shown in Figure 7. The *weight of a path* is the product of the weights of its edges.



**Figure 7. The detection graph of a binary association**

The identification graph is used for identifying causes for non finite satisfiability of a class diagram. Cycles whose weight is less than 1, termed *critical cycles*, point on non finite satisfiability. Moreover, a critical cycle singles out a non-finitely satisfiable set of multiplicity constraints. Figure 8b shows the identification graph for Figure 8a, and a critical cycle, that singles out the unary *predecessor-successor* association as the finite-satisfiability cause.



**Figure 8. An identification graph with a critical cycle**

Maraee et al. (2008) have extended these results to apply to class diagrams with class hierarchy, qualifier and association class constraints. Detecting finite satisfiability in presence of *GS* constraints is still an open issue.

Hartman (2001) handles finite satisfiability in diagrams with binary multiplicity constraints from all three aspects of problem detection, cause identification and advice. For cause identification, he suggests an algorithm for determining all non-finitely satisfiable classes, based on identification of *minimal distances* in critical cycles. His advice approach involves automatic corrections.

### 3.3 Reasoning about Class Diagrams with Constraints

The expressivity of class diagrams is limited to class level interaction and constraints. The Object Constraint Language OCL (OMG-OCL, 2004; Warmer & Kleppe, 2003) is intended to extend a UML model (mainly class diagrams) with symbolic constraints. Cabot et al. (2008) describe a CSP-based tool for reasoning about finite satisfiability of class diagrams that are

extended with OCL constraints. USE is a tool for validation of UML/OCL models (Gogolla et al., 2001; Richters & Gogolla, 2000). Alloy is a Z based tool for automated analysis of object structured software specification (Jackson, 2002). Together with the recent UML2Alloy tool (Anastasakis et al., 2007), it provides a UML/OCL analysis tool. Most UML/OCL tools do not separate reasoning about visual constraints from symbolic OCL constraints. Therefore, since OCL, as an unrestricted first order logic language, is undecidable, Cabot et al. (2008) and Jackson (2002) perform incomplete bounded verification.

## 4. Explaining and Repairing Correctness Problems Using a Pattern-Based Approach

The goal of a model development tool is to help the designer in developing a correct and high quality model. For this purpose, the tool should point at design problems, explain their cause, and suggest possible repairs. The detection and cause identification methods, surveyed in the previous section, do not function as advice and explanatory tools. Consider, for example, the non finite satisfiability problem in Figure 1. The identification graph method (Maraee et al., 2008) detects the critical cycle *CatalyzedReaction, Enzyme, Protein, Molecule, Chemical, Reaction, CatalyzedReaction*. But the critical status of this cycle can be caused by several interactions of constraints:

1. The *Enzyme-Chemical* class hierarchy and the multiplicity constraints in the rest of the critical cycle.
2. The *Reaction-CatalyzedReaction* class hierarchy and the multiplicity constraints in the rest of the critical cycle.
3. The multiplicity constraints in the critical cycle.

The reason lies in the second interaction: The intended direction of class hierarchy, from the subclass *CatalyzedReaction* to the super-class *Reaction* is reversed. This final conclusion is domain dependent, and can be made by the designer, based on personal knowledge, or by consulting appropriate domain ontologies. Therefore, a desirable approach seems to involve proposing possible explanations, and letting the designer take the final repair decision (in contrary to the approach of Hartman (2001)).

We suggest bridging the gap between current correctness handling methods and desirable explanations and advice, by using a pattern based approach. This idea is influenced from the design patterns and anti-patterns paradigms, where patterns function as advices for solving typical problems that can occur in various contexts. Design patterns fulfill an educational role: Awareness to design patterns yields better solutions. In analogy, patterns of correctness problems characterize typical situations in which correctness problems arise, analyze the causes, and suggest possible solutions. Their educational role is to increase the awareness of designers to avoid contradictory constraints that cause correctness problems.

We present *correctness patterns*, which characterize typical cases of erroneous design. Each pattern describes a correctness problem, caused by problematic interaction of constraints, and can be identified by characteristic structures within a class diagram. A pattern is also associated with advices for repairing the identified problem. Each pattern includes proofs that justify the identification structure and the repair advices.

The presented patterns characterize four correctness problem types – *inconsistency*, *non-finite satisfiability*, *redundancy* and *incomplete design* – which represent two aspects of *correctness*. The first two are problems of *formal correctness*, while the last two are problems of *design quality*, which is another form of correctness: A low quality design is formally correct, but does not meet criteria of desirable design. In this paper two patterns are described in detail, and few others are presented in a condensed format, including only concrete examples.

#### **4.1 Patterns of Inconsistency and Non-Finite Satisfiability Problems**

Inconsistency is caused by contradictory constraints that cannot be simultaneously satisfied. Finite satisfiability problems are caused by cycles of conflicting multiplicity constraints. The cycles might involve other constraints, like class hierarchies, generalization sets, association classes, inter-association and OCL constraints.

##### **Lack of Finite Satisfiability patterns:**

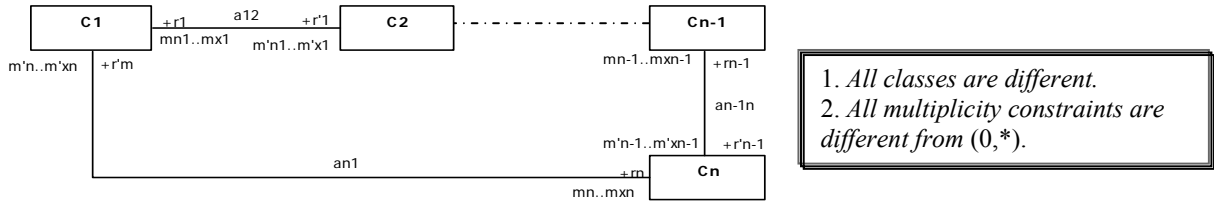
###### ***1. Multiplicity constraint interaction:***

**Pattern Name:** Pure Multiplicity Cycle (PMC)

**Pattern Description:** A cycle of associations with multiplicity constraints might introduce a finite satisfiability problem.

**Pattern Identification Structure:**

A minimal association cycle in which all multiplicity constraints are different from (0, \*). Figure 9 shows the pattern’s identification structure.



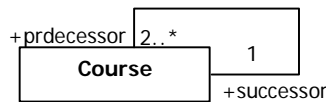
**Figure 9: The PMC Pattern**

The identification structure can be symbolically described using a regular expressions notation (\* denotes unbounded repetition, \*n..m denotes bounded repetition, + denotes alternatives):

$$[C_{i \min_i, \max_i} \text{---} \min'_i, \max'_i C_{i+1}]^{*1..(n-1)} \min'_n, \max'_n \text{---} \min'_n, \max'_n C_1$$

**Concrete Example:**

Figure 10 is an example of a minimal association cycle that causes a finite satisfiability problem. Each course has a single successor and at least two predecessors. Therefore, if the number of courses in a legal instance is C, and the number of predecessor-successor links is D, then D must satisfy:  $D = C * I$  and  $D \geq C * 2$ , implying the inequality:  $C \geq C * 2$ , that can be satisfied only by an empty or an infinite extension of the class *Course*.



**Figure 10: Non-finite satisfiability due to multiplicity constraint interaction**

**Pattern justification:**

The example above proves that an association cycle without a (0,\*) multiplicity constraint can cause a finite satisfiability problem. We still need to show that the conditions of minimality of the cycle and the “no (0,\*) multiplicity constraint” are necessary, i.e., a minimal association cycle with a (0,\*) multiplicity constraint cannot cause a finite satisfiability problem, and dropping the minimality requirement might miss some problems. The proof relies on properties of the identification graph of the cycle (see Figure 8, Lenzerini & Nobili (1990), and Maraee et al. (2008)). It shows that if a minimal association cycle AC includes a (0,\*) multiplicity

constraint, then its identification graph  $IG_{AC}$  does not include a critical cycle. Moreover, if the cycle is not minimal, then  $IG_{AC}$  might include a critical cycle even if the  $AC$  includes a  $(0,*)$  multiplicity constraint.

**Proposition:** A cycle in  $IG_{AC}$  that includes an edge for a 0 minimum constraint or a \* maximum constraint is not critical.

**P proof:** The weight of such edges in  $IG_{AC}$  is  $\infty$ .

**Proposition:** Let  $AC = [C_{i \min_i, \max_i} \text{---} \min'_i, \max'_i C_{i+1}]^{*1..(n-1)} \min_n, \max_n \text{---} \min'_n, \max'_n C_1$  be a minimal association cycle, such that its identification graph  $IG_{AC}$  includes a critical cycle. Let  $IG_{cycle}$  be a minimal critical cycle in  $IG_{AC}$ , i.e., a critical cycle that cannot be pruned. Then, the edges of  $IG_{cycle}$  correspond to alternating minimum-maximum constraints in  $AC$ :  $\max'_1, \min_1, \max'_2, \min_2, \dots, \min_n$ , or  $\max_n, \min'_n, \max_{n-1}, \min'_{n-1}, \dots, \min'_1$ .

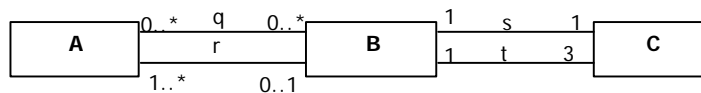
**Proof (sketched):**

$IG_{cycle}$  cannot include successive edges for a multiplicity constraint  $(\min_i, \max_i)$ , since such edges create a cycle whose weight is greater than 1, and therefore can be pruned, in contradiction to the minimality of  $IG_{cycle}$ . Therefore,  $IG_{cycle}$  passes through different associations in a cycle of  $AC$ . Since  $AC$  is minimal (does not include inner cycles),  $IG_{cycle}$  passes through all associations in  $AC$ , implying its claimed structure.

**Conclusion 1:** A minimal association cycle  $AC$  that includes a  $(0,*)$  constraint cannot create a finite satisfiability problem (since every cycle in  $IG_{AC}$ , that passes through all associations of  $AC$ , includes an edge, either for the 0 or for the \* constraints, and therefore cannot be critical).

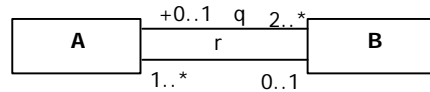
**Conclusion 2:** The following cases of association cycles might introduce a finite satisfiability problem:

1. A non-minimal association cycle that includes a  $(0,*)$  constraint:



**Figure 11: A non-minimal association cycle that includes a  $(0,*)$  constraint**

2. An association cycle with a multiplicity constraint (0, n) or (n, ∞), where  $n \neq 0, \infty$ :



**Figure 12: Non-finite satisfiability due to multiplicity constraint interaction**

The class diagram in Figure 10 also shows that case.

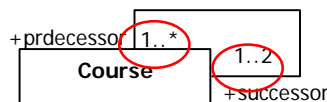
**Conclusion 3:** A minimal association cycle  $AC$  without (0,\*) creates a finite satisfiability problem if and only if one of the following conditions holds:

$$max'_1 \cdot max'_2 \cdot \dots \cdot max'_n \geq min_1 \cdot min_2 \cdot \dots \cdot min_n$$

$$max_n \cdot max_{n-1} \cdot \dots \cdot max_1 \geq min'_n \cdot min'_{n-1} \cdot \dots \cdot min'_1$$

**Pattern verification:** This pattern characterizes a necessary but not sufficient condition on the multiplicity constraints in an association cycle. In order to find out whether a PMC causes a finite satisfiability problem, there is a need to check the conditions in conclusion 3.

**Repair advice:** Consider increasing a maximum constraint or decreasing a minimum constraint, such that one of the conditions in Conclusion 3 holds. For example, in Figure 10, the minimum predecessor requirement for a course can be decreased, or the maximum successor requirement can be increased as shown in Figure 13.



**Figure 13: A repaired class diagram of Figure 10**

2. **Interaction of multiplicity and class hierarchy constraints:**

**Pattern Name:** Multiplicity Hierarchy Cycle (MHC).

**Pattern Description:** A cycle of associations with multiplicity constraints and class hierarchy constraints might introduce a finite satisfiability problem

**Pattern Identification Structure:**

A minimal cycle of associations and class hierarchy constraints, in which all multiplicity constraints are different from (0,\*). Figure 9 shows the pattern's identification structure.

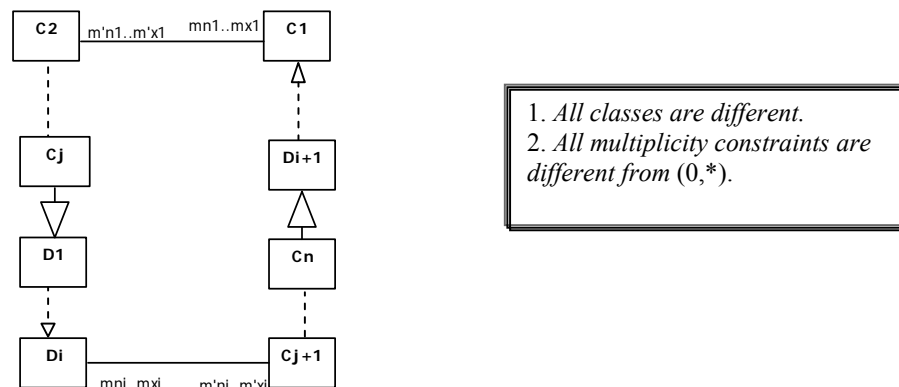


Figure 14: The MHC Pattern

The identification structure can be symbolically described using a regular expressions notation:

$$[(C_i \text{ min}_i, \text{max}_i \text{ --- min}'_i, \text{max}'_i C_{i+1}) + (C_i < C_{i+1}) + (C_i > C_{i+1})]^{*1..(n-1)} \text{ --- min}'_n, \text{max}'_n C_1$$

**Concrete Example:**

Figure 15a presents a multiplicity constraint cycle that involves class *Graduate*, which is a subclass of class *Academic*, and whose instances must be related to *Academic* instances. Therefore, assuming that *G* and *A* are the number of graduates and academics, respectively, the number of *student-advisor* links in every legal instance must be both, *G*\*1 and *A*\*2, implying *G* = *A*\*2. In addition, the extensions of *Graduate* and *Academic* must satisfy *G* ≤ *A*, since *Graduate* is a subclass *Academic*. These constraints can be satisfied only by empty or infinite extensions. Figure 15b presents a finite satisfiability problem which is a reduced version of the finite satisfiability problem in Figure 1.

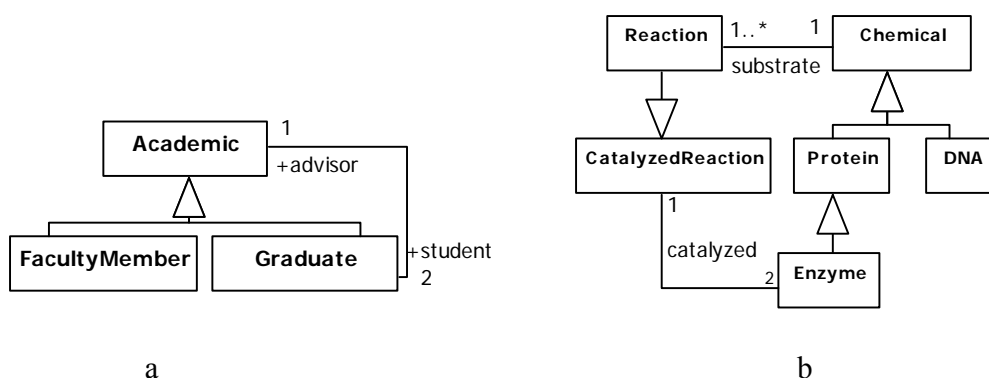


Figure 15: Non-finite satisfiability due to multiplicity and class hierarchy constraints

### Pattern justification:

The above examples show that an association–class-hierarchy cycle can cause a finite satisfiability problem. It is left to show that the minimality of the cycle and the “no (0,\*) multiplicity constraint” are necessary conditions. As in the previous pattern, the proof relies on the identification graph of the cycle (Maraee et al., 2008). The identification graph is constructed by first replacing every class hierarchy constraint in the cycle by an association with multiplicity constraints (0, 1) and (1, 1) on the sub-class and super-class sides, respectively. This replacement creates a regular association cycle, for which the identification graph is constructed, following Lenzerini & Nobili (1990). For that cycle, the PMC pattern already justifies the “no (0,\*) multiplicity constraint” requirement. Since the class hierarchy to association translation does not insert a (0,\*) multiplicity constraint, we conclude that if the original association cycle is minimal and does not include a (0,\*) multiplicity constraint, the cycle might cause a finite satisfiability problem, while the presence of a (0,\*) multiplicity constraint guarantees that no finite satisfiability problem can be caused by this cycle.

### Pattern verification:

In order to find out whether an MHC cause a finite satisfiability problem, there is there is a need to check the conditions in conclusion 3.

### Repair advice:

Consider revising the multiplicity constraints, as in the PMC repair advice, or switching the direction of a class hierarchy constraint in the cycle. The latter can repair the finite satisfiability problem since it switches the positions of the (0,1) and the (1,1) multiplicity constraints, created for a class hierarchy constraint. For example, in Figure 16, switching the direction of the *CatalyzedReaction-Enzyme* hierarchy, which indeed is correct, removes the finite satisfiability problem.

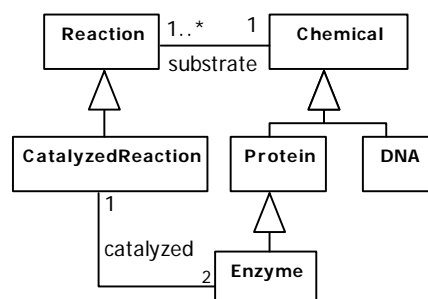


Figure 16: A repaired class diagram

### 3. Interaction of multiplicity constraints via an association class constraint (condensed):

Figure 17 presents an association class *Contract* that is constrained by contradictory multiplicity constraints. In every legal instance, the number  $C$  of contracts is as twice as the number of employees,  $E$ , since every employee is linked to two departments. Yet, the number of contracts is equal to the number of employees, as dictated by the *manages* 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.

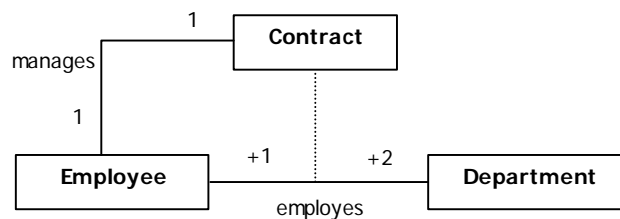


Figure 17: Non-finite satisfiability due to multiplicity and association class constraints

### 4. Interaction of multiplicity and Generalization Set constraints (condensed):

In Figure 18 the  $\{disjoint, incomplete\}$  constraint suggests that the *DNASegment* extension properly includes the *Gene*, *Chromosome* and the *Genome* extensions. Yet, *DNASegment* instances are mapped in a 1:1 manner to *Genome* instances, implying that the sets have the same size. The only solution for proper set inclusion with equal size is that the sets are either empty or infinite.

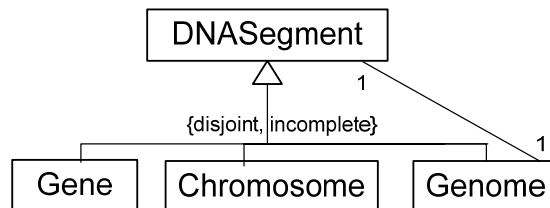
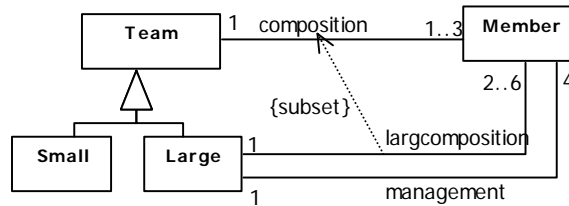


Figure 18: Non-finite satisfiability due to multiplicity and generalization set constraints

### 5. Interaction of multiplicity and inter-association hierarchy constraints (condensed):

In Figure 19 the subset constraint between the *largecomposition* and the *composition* associations tightens the multiplicity constraint of *Member* in *largecomposition* into 2..3, which 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  $L*2 \leq LC \leq L*3$ ,  $LC = M*1$  imply  $L*2 \leq M \leq L*3$ . Replacing  $M$  by  $L*4$  yields the inequality  $L*2 \leq L*4 \leq L*3$ , that can be satisfied only by either empty or infinite extensions for *Large*, and *Member*.

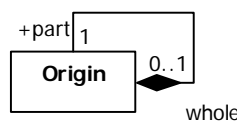


**Figure 19: Non-finite satisfiability due to class hierarchy, association hierarchy and multiplicity constraints**

**6. Lack of finite satisfiability due to acyclic structure of aggregation and composition (condensed):**

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 20 describes a class whose instances aggregate instances of the same class.

The whole-part association defines a one-to-one function from the set *Origin* to itself, which maps an element of *Origin* to its single part. If the extension of *Origin* is a finite non-empty set, the mapping must be cyclic. Therefore, *Origin* might have only empty or infinite extensions.



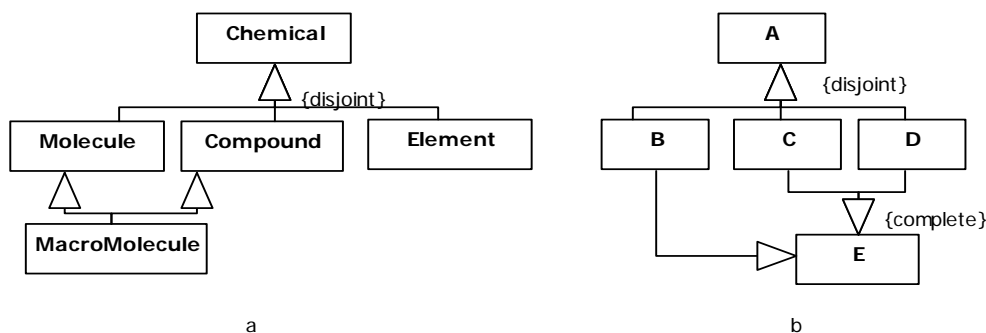
**Figure 20: Unsatisfiability due to the asymmetry of aggregation**

Wahler et al. (2009) present the No Cyclic Dependency pattern which prevents cyclic links between objects of classes<sup>6</sup> within a path of associations. The authors present a sufficient condition for preserving the finite satisfiability of a class diagram after enhancing a path of associations in the class diagram with No Cyclic Dependency pattern. The condition states there must be one association end in the path whose lower multiplicity bound is zero and, in addition there is one association end whose opposite end also has a lower multiplicity bound of zero. Similar to the above aggregation pattern, it can be shown that if such condition does not hold, there can be only a non finite satisfiable instance of this path. Based on this observation, a possible repair that we suggest to aggregation pattern is to select two association ends in the recognized pattern and change to zero the lower multiplicity bound these association ends.

### **Inconsistency patterns (condensed):**

#### **1. *Contradictory generalization set constraints:***

Figure 21(a) presents the *diamond class hierarchy*, in which the *disjoint* constraint enforces the *MacroMolecule* class to be empty. In Figure 21(b), the interaction of the *disjoint* and the *complete* constraints forces class *B* to be empty, since an instance of *E* must be an instance of *C* or *D*, which are disjoint from *B*.

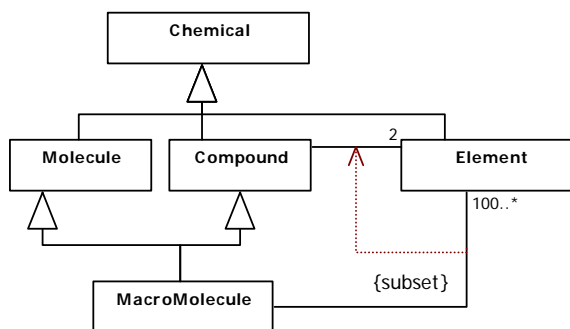


**Figure 21: Inconsistency due to generalization-set constraints**

#### **2. *Interaction of multiplicity and inter-association hierarchy constraints:***

<sup>6</sup> Constraint patterns are parameterized pattern expressions that can be instantiated to a specific pattern. For more details we refer the readers to Ackermann & Turowski (2006) and Wahler (2008).

In Figure 22 the subset constraint between the *MacroMolecule-Element* and the *Compound-Element* associations implies contradiction between the multiplicity constraints of class *Element* within these associations.



**Figure 22: Inconsistency due to multiplicity and class hierarchy constraints**

#### 4.2 Patterns of Redundant and Incomplete Design (condensed)

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. Values that cannot be realized in multiplicity constraints are redundant. In the first case, one of the equivalent elements can be removed. In the latter case, the multiplicity value range can be tightened.

Incomplete design means that implied constraints do not appear in the diagram. It reflects lack of awareness on the designer's part, and therefore shows low design quality. In some cases, incomplete design can be interpreted as syntactically erroneous, and be rejected by modeling tools. Some problems of class diagram redundancy and implicit consequences are identified and discussed by Berardi et al. (2005) and Costal & Gomez (2006).

##### 1. **Redundancy due to equivalent classes:**

Class redundancy occurs mainly due to class hierarchy. In Figure 23 the class *D* extension is a subset of *A* extension due to the class hierarchy constraint, and the sets have the same size, since

$A$  instances are mapped in a 1:1 manner to  $D$  instances. If the sets are finite, they must be equal, and therefore the  $D$  class is redundant and can be removed.

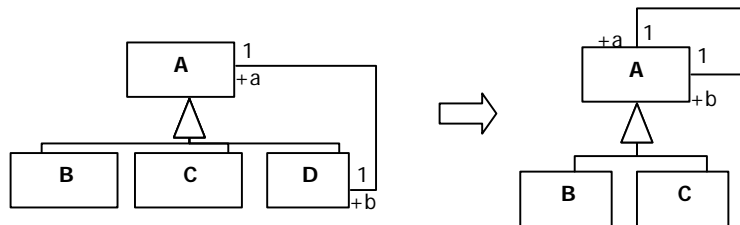


Figure 23: Redundancy due to equivalent classes

## 2. Redundancy due to equivalent associations

This case is the same as the previous. Association equivalence occurs due to hierarchy of association classes. In Figure 24,  $R$  and  $Q$  are associations that constrain the association classes,  $P$  and  $S$ , respectively, and are also constrained by a 1:1 multiplicity constraint. Therefore,  $P$  and  $S$  have equal extensions in all legal instances, and associations  $R$  and  $Q$ , and one of the association classes  $P$  and  $S$  are redundant.

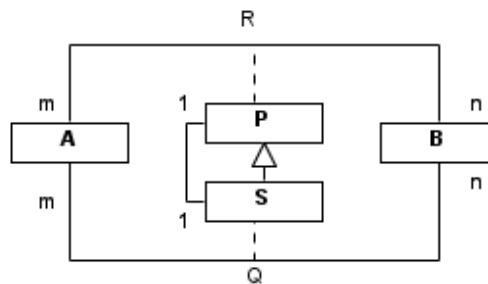


Figure 24. Redundancy due to equivalent associations

## 3. Redundancy of multiplicity constraint values:

In Figure 25 the multiplicity constraint  $1..3$  of the *Member* class in *Composition*, makes its multiplicity constraint  $1..*$  in *LargeComposition* too loose, as the maximal multiplicity is 3.

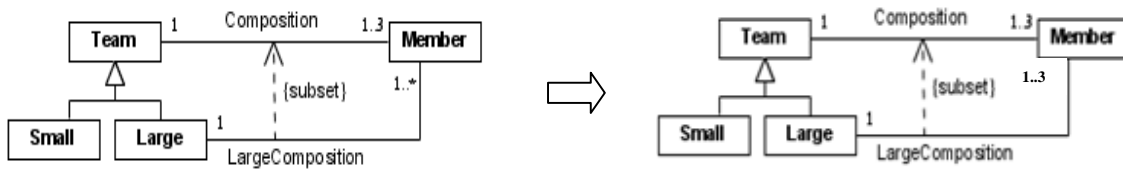


Figure 25: Redundancy due to cardinality constraints

#### 4. Incomplete design:

Figure 26 demonstrates a design improvement transformation, based on detection of incomplete semantic constraint specification. The problem is that association hierarchy can apply only to class pairs that admit class hierarchy constraints. Instead of rejecting the diagram, a modeling tool might adopt a credulous approach, by inferring a repair to the detected problem.

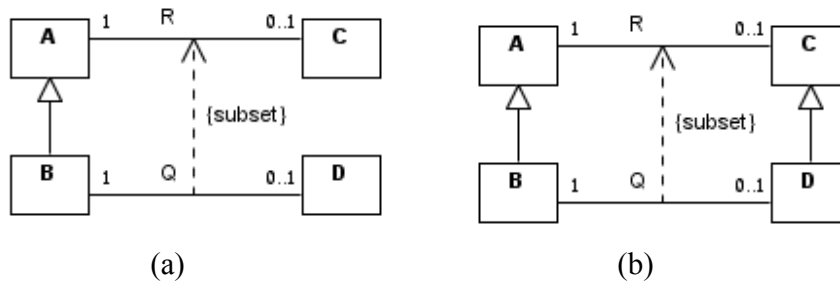


Figure 26: Design improvement due to incomplete hierarchy structure class equivalence

## 5. Summary

In this paper we have analyzed correctness problems in UML class diagrams, surveyed existing approaches for handling correctness of UML class diagrams, and proposed a pattern-based approach for identifying correctness problems and for providing explanations and repair advices. The proposed approach aims at bridging the gap between current correctness handling methods and desirable explanations and advice. Our work is motivated by the believe that correctness management is essential for supporting advanced IDE and CASE tools in all application areas of UML class diagrams. In addition, the emerging model driven development approach requires reliable models that are equipped with powerful reasoning capabilities for assuring high quality models.

We are currently working on an on-line catalog of the various correctness patterns to be served for accumulating the knowledge regarding the various patterns to be implemented, as well as for educational purposes. We intend to further develop the pattern-based approach presented here, examine its applicability (complexity wise), and combine it within an implementation of existing cause identification methods.

We envision that the next generation of modeling tools will apply some correctness management facilities. Tools will employ a mixture of reasoning methods, applying simple scalable methods in an incremental way, and resorting to heavy translation-based reasoning when other methods fail.

## References

- Ackermann, J., & Turowski, K. (2006). A Library of OCL Specification Patterns to Simplify Behavioral Specification of Software Components. In K. Pohl (Ed.), *Advanced Information Systems Engineering: 18th International Conference, CAiSE 2006, Luxembourg* (pp. 255-269). Springer Berlin / Heidelberg
- Ali, A. B., Boufares, F. & Abdellatif, A. (2006). On the global coherence of integrity constraints in UML class diagrams. *Proceeding of the 24th IASTED international conference on Database and applications* (pp. 109-114). ACTA Press, Anaheim, CA.
- Anastasakis, K., Bordbar, B., Georg, & G., Ray, I. (2007). UML2Alloy: A Challenging Model Transformation. In G. Engels, B. Opdyke, D. C. Schmidt, & F. Weil (Eds.), *Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville, USA* (pp. 436-450). Springer Berlin / Heidelberg.
- Artale A., Calvanese D., Kontchakov R., Ryzhikov V., & Zakharyashev M. (2007). Complexity of Reasoning over Entity-Relationship Models. *Proc. of the 20th Int. Workshop on Description Logic (DL 2007)*, 250, (pp. 163-170).
- Balaban, M. & Maraee, A. (2006). Consistency of UML Class Diagrams with Hierarchy Constraints. In O. Etzion, T. Kuflik & A. Motro (Eds), *Next Generation Information Technologies and Systems: 6th International Conference, NGITS 2006* (pp. 71-82). Springer Berlin / Heidelberg.
- Balaban, M. & Shoval, P. (2002). MEER -- An EER Model Enhanced with Structure Methods. *Information Systems*, 27, 245-275.
- Berardi, D., Calvanese, D., & De Giacomo, G. (2005). Reasoning on UML class diagrams. *Artificial Intelligence*, 168 (1), 70-118.
- Blaha, M., Premerlani, W., & Shen, H. (1994). Converting OO Models Into RDBMS Schema. *IEEE Software*, 11 (3), 28-39.
- Boufares, F. & Bennaceur, H. (2004). Consistency Problems in ER-schemas for Database Systems. *Information Sciences*, 163 (4), 263-274.

- Cabot, J. and Clariso, R. & Riera, D. (2008). Verification of UML OCL Class Diagrams using Constraint Programming, In *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)*.
- Cadoli, M., Calvanese, D., De Giacomo, G. & Mancini, T. (2004). Finite satisfiability of UML class diagrams by constraint programming. In M. Wallace (Ed.), *The CP-04 Workshop on CSP Techniques with Immediate Application*.
- Calvanese, D., & Lenzerini, M. (1994). On the Interaction between ISA and Cardinality Constraints. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, (pp. 204-213). IEEE Computer Society, Washington, DC.
- Calvanese, D., De Giacomo, G., Lenzerini, M., Nardi, D., & Rosati, R. (1998). Description Logic Framework for Information Integration. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR'98)* (pp. 2-13). Morgan Kaufman, Los Altos, CA.
- Carsten L., Ulrike S., & Lidia T. (2005). The Complexity of Finite Model Reasoning in Description Logics. *Information and Computation*, 199, 132-171.
- Costal, D., & Gomez, C., (2006). On the Use of Association Redefinition in UML Class Diagrams, Conceptual Modeling. In D. W. Embley, A. Olivé, S. Ram (Eds), *Conceptual Modeling - ER 2006: 25th International Conference on Conceptual Modeling* (pp. 513-527). 2006. Springer Berlin / Heidelberg.
- Cranefield, S. & Purvis, M. (1999). UML as an ontology modeling language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*.
- Cranefield, S. (2001). Networked knowledge representation and exchange using UML and RDF. *Journal of Digital Information*, 1(8).
- Dobing, B. & Parsons, J. (2006). How UML is used. *Communication of the ACM*, 49 (5), 109-113.
- Eclipse, <http://eclipse-plugins.info/>, 2006.
- Falkovych, K., Sabou, M., & Stuckenschmidt H. (2003). UML for the Semantic Web: Transformation-Based Approaches. In M. Klien (Ed.), *Knowledge Transformation for the Semantic Web*, (92-106). IOS Press; illustrated edition.
- Gasevic, D., Djuric, D., Devedzic, V., & Damjanovi, V. (2004). Converting UML to OWL ontologies. In *Proceedings of the 13th international World Wide Web Conference on Alternate Track Papers & Posters* (pp. 488-489).
- Gogolla, M., Bohling, J. & Richters, M. (2001). Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4(4), 386-398.
- Haarslev, V., & Möller R. (2001). RACER System Description, In *Proceedings of the International Joint Conference on Automated Reasoning, IJCAR'2001*, (pp. 701-705).

- Hartmann, S. (2001). Coping with Inconsistent Constraint Specifications. In *Proceedings of International Conference on Conceptual Modeling - ER 2001*, (pp. 241-255). Springer-Verlag.
- Heckel, R. & Voigt H. (2003). Towards consistency of web service architectures. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics, and Informatics*.
- Horrocks, I. (1998). The FaCT system. In *Proceedings of the 2nd International Conference on Analytic Tableaux and Related Methods (TABLEAUX'98)*, (pp. 307-312). Springer-Verlag.
- Huzar, Z., Kuzniarz, L., Reggio, G., & Sourrouille, J-L. (2004). Consistency Problems in UML-Based Software Development. *UML Satellite Activities*, 1-12.
- Jackson, D. (2002). Alloy: A New Technology for Software Modelling. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag.
- Jackson, D., & Rinard, M. (2004). Software analysis: a roadmap, in *Proceedings of the Conference on the Future of Software Engineering*, (pp. 133-145). Springer-Verlag.
- Kabilan, V., & Johannesson, P. (2004). UML for Ontology Modeling and Interoperability, *CAiSE Workshops (3)*, 349-354.
- Karsai, G., Maroti, M., Ledeczi, A., Gray, J., & Sztipanovits J. (2003). Composition and Cloning in Modeling and Meta-Modeling. *IEEE Transactions on Control System Technology*, 12 (2), 263- 278.
- Kogut, P. A., Cranefield, S., Hart, L., Dutra, M., Baclawski, K., Kokar, M. M. and Smith, J. E. (2002). UML for Ontology Development, *The Knowledge Engineering Review*, 17 (1), 61 – 64.
- Kozlenkov, A. & Zisman, A. (2004). Discovering, Recording, and Handling Inconsistencies in Software Specifications. *International Journal of Computer and Information Science*, 5(2), 89-108.
- Lange, C.F.J., Chaudron, M.R.V., & Muskens J. (2006). In Practice: UML Software Architecture and Design Description. *IEEE Software*, 23 (2), 40-46.
- Lenzerini, M. & Nobili, P. (1990). On the satisfiability of dependency constraints in entity-relationship schemata, *Information Systems*, 15 (4), 453-461.
- Maraee A. & Balaban, M., Efficient Decision of Consistency in UML Diagrams with Constrained Generalization Sets. In *Proceedings of the first Workshop on Quality in Modeling*.
- Maraee, A. (2007). *Finite Satisfiability Problems in UML Class Diagram*. Unpublished Master thesis, Ben-Gurion University of the Negev.
- Maraee, A., & Balaban, M. (2007) Efficient Reasoning About Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In D. Akehurst, R. Vogel, & R. Paige (Eds), *Model Driven Architecture - Foundations and Applications: Third European Conference*, (pp. 7-31). Springer-Verlag.

- Maraee, A., & Balaban, M. (2008). Efficient Recognition of Finite Satisfiability in UML Class Diagrams: Strengthening by Propagation of Disjoint Constraints. In *the Second International Conference on Model Based Systems Engineering*.
- Maraee, A., Makarenkov, V., & Balaban, M. (2008). *Efficient Recognition and Detection of Finite Satisfiability Problems in UML Class Diagrams: Handling Constrained Generalization Sets, Qualifiers and Association Class Constraints*. Paper presented at the 1st International Workshop on Model co-evolution and consistency management.
- Martin, R.C. (2006). *UML for Java™ Programmers*. Prentice Hall.
- OMG-OCL, UML 2.0 OCL Specification, 2004.
- OMG-UML, UML 2.0 Superstructure, 2006.
- Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., & Wroe, C. (2004) OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. In *Proceedings of the European Conference on Knowledge Acquisition (pp. 63-81)*. Springer-Verlag.
- Richters, M. & Gogolla, M. (2000). Validating UML Models and OCL Constraints. In *Proceedings of UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference*.
- Rumbaugh, J., Jacobson, I. & Booch, G. (2005). *The Unified Modeling Language Reference Manual*, Second Edition, Addison-Wesley, 2005.
- Satoh, K., Kaneiwa, K. & Uno, T. (2006). Contradiction Finding and Minimal Recovery for UML Class Diagrams. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*.
- Schild, K. (1992). From Terminological Logics to Modal Logics. *Description Logics*, 101-104.
- Stahl, T., Voelter, M., Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*, Wiley.
- Sunye, G., Pollet, D., Le Taraon, Y. & Jezkel J.-M. (2001). Refactoring UML models. In *Proceedings of UML 2001 (pp. 134-148)*.
- Thalheim, B. (1993). Fundamentals of Entity-Relationship Modeling. *Annals Mathematics and Artificial Intelligence*, 7, 197-256,
- Timm, J.T.E. & Gannod, G., (2005). A model-driven approach for specifying semantic web services. In *Proceeding of International Conference of Web Services (pp. 313–320)*.
- Tsarkov, D. & Horrocks, I. (2006). FaCT++ description logic reasoner: System description. In *Proceedings of the International Joint Conference on Automated Reasoning (pp. 292-297)*.
- Unhelkar, B. (2005). *Verification and Validation for Quality of UML 2.0 Models*, Addison Wesley.

- Wahler, M. (2008). Using Patterns to Develop Consistent Design Constraints. *PhD thesis*, ETH Zurich, Switzerland, No. 17643, 2008.
- Wahler, M., Basin, D., Brucker, A. D. & Koehler, J. (2009). Efficient Analysis of Pattern-Based Constraint Specifications, *Software and Systems Modeling*, DOI: 10.1007/s10270-009-0123-6 2009.
- Warmer, J. & Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley.