Unsatisfiability Reasoning in ORM Conceptual Schemes

Mustafa Jarrar[1] Stijn Heymans[2]

[1] STAR Lab, Vrije Universiteit Brussel, Belgium, mjarrar@vub.ac.be
[2]TINF, Vrije Universiteit Brussel, Belgium, sheymans@vub.ac.be

Abstract. ORM (Object-Role Modeling) is a rich and well-known conceptual modeling method. As ORM has a formal semantics, reasoning tasks such as satisfiability checking of an ORM schema naturally arise. Satisfiability checking allows a developer to automatically detect contradicting constraints. However, no *complete* satisfiability checker is known for ORM. In this paper, we revisit existing patterns from literature that indicate unsatisfiability of ORM schemes i.e., schemes that cannot be populated, and we propose refinements as well as additions for them. Although this does not yield a complete procedure – there may be ORM schemes passing the pattern checks while containing unsatisfiable roles – it yields an efficient and easy to implement detection mechanism (specially in interactive modeling tools) for the most common conceptual modeling mistakes.

1 Introduction

ORM (Object-Role Modeling) is a conceptual modeling approach and the historic successor of the NIAM (Natural-language Information Analysis Method) [VB82]. ORM schemes can be translated into pseudo natural language statements. The graphical representation and the translation into pseudo natural language make it a lot easier, also for non-computer scientists, to create, check and adapt the knowledge about the Universe of Discourse (UoD) needed in an information system.

Although ORM was originally developed as a database modeling approach, it has been also successfully reused in other conceptual modeling scenarios, such as *ontology modeling* [J05], business rule modeling [H97,N99,DJM02a], XML-Schema conceptual design [BGH99], etc. Hence, we shall regard an ORM schema, in this paper, as a general conceptual model independently of a certain modeling scenario or domain[1].

ORM has a well-defined formal semantics (see e.g. [H89,BHW91,T96,TM95]). This formal semantics naturally leads to the question of satisfiability checking, i.e. given a concept/role in a schema, is there a model (an interpretation/population of the schema that satisfies all constraints) such that the concept/role has a non-empty population. From a practical perspective, such reasoning procedures can help the developer in analyzing the *validity* of the constructed schema for the domain. In particular, it allows to detect concepts and roles in a schema that always have an empty population,

[1] We sometimes interchange the term “ORM schema” with the term “axiomatization” to refer to the same thing.

symptoms of a faulty model: there are too many constraints or constraints are too harsh[2].

For example, consider Fig. 1, stating that Students and Employees are types of Persons where no Student can be an Employee (and vice versa), and a PhD Student is both a Student and an Employee. Thus, the PhDStudent type cannot be populated. Otherwise, a PhD Student would be both a student and an employee which contradicts with the fact that Student and Employee need to be disjoint types (by the exclusion constraint). Although there are types in the schema in Fig. 1 that cannot be satisfied, there is a formal model satisfying the global schema: e.g. let PhDStudent have an empty population, Student and Employee disjoint populations, and Person some superset of the union of the populations of Student and Employee.

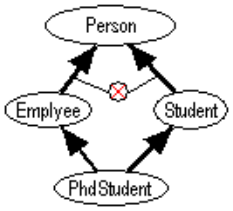


Fig. 1. Unsatisfiability of ORM schema.

Types of Satisfiability. Formally, we distinguish between three types of satisfiability of an ORM schema [BHW91]. First, *schema satisfiability checking* of an ORM schema is checking whether there exists a model of the schema (or less abstract, some population for the schema) as a whole. A satisfiable ORM schema does not need to satisfy any concepts or roles per se, as exemplified in Fig. 1. The only condition is that all constraints are satisfied by the (possibly empty) populations. Second, *concept satisfiability checking* amounts to checking whether *all* concepts (i.e. object-types) are satisfied (can be populated) by a model (by a population) of the schema. Concept satisfiability is thus stronger than schema satisfiability as a model of the schema that satisfies all concepts is, by definition, also a model of the schema. Finally, *role satisfiability checking* amounts to checking whether there exists a model of the schema that satisfies (populates) *all* roles in the schema. This is the strongest form of satisfiability checking as it implies concept satisfiability: if a role is satisfied, the corresponding concept that plays the role is also satisfied. Given these implications (*role then concept then schema satisfiable)*, we refer to role satisfiability as *strong satisfiability* and to *schema satisfiability* as *weak satisfiability*.

In this context, we are particularly interested in strong satisfiability: checking whether *all* roles in the schema are satisfiable: since a weakly satisfiable model may contain empty roles, problems with contradictory constraints are not necessarily detected. Note that if the schema does not contain roles we will also look at concept satisfiability.

In this paper, we refine, optimize, and re-represent existing ORM "formation rules" found formed in [H89,DMV,BHW91], as well as introduce new ones for detecting unsatisfiable roles (we call it *unsatisfiability constraint patterns*). Furthermore, we

[2] We assume the UoD itself is consistent, such that faults in the model have their origin in the modeling and not in the UoD.

indicate how a significant part of these existing rules are not aimed at detecting unsatisfiability but are guidelines for good modeling, e.g., to avoid redundant/implied constraints.

Our patterns approach for detecting unsatisfiable models is motivated by the needs that *unsatisfiability procedures should be easy to implement and applicable in interactive modeling.* The latter requirement enforces that the procedures should be fast as well. We shall come back to this motivation in Sec. 4, and illustrate how our approach can be applied in *interactive* ontology modeling tools, which indeed help ontology builders to quickly detect unsatisfiability in the early phases of ontology modeling. Our experience in applying this approach for the development of a customer complaint ontology (developed by 10s of lawyers) will be reported in Sec. 4 as well.

Although our approach covers the most common unsatisfiability cases in practice, it cannot be complete, i.e., there are ORM schemas that are not strongly satisfiable but will fail to be detected by our patters. This is because of the general problem of determining consistency for all possible constraint patterns in ORM is undecidable [H97], e.g., the use of transitivity combined with frequency constraints is problematic with regards to decidability. In Sec. 4, we shall discuss and compare our patterns approach with the work that we have done on mapping (large part of) ORM into description logics (DLs) [JF05]. We shall discuss that the patterns approach (which we present in this paper) offers fast and easy to implement and understand reasoning mechanism specially for interactive modeling. We believe that both approaches complement each other.

The remainder of the paper is organized as follows. In Sec. 2, we introduce 9 patterns for detecting unsatisfiability of ORM models. Section 3 discusses related work, Sec. 4 illustrates the implementation of the patterns, and Sec. 5 contains discussion, conclusions and directions for future work.

2 Unsatisfiability Patterns in ORM Conceptual Schemes

In this section, we present and discuss 9 patterns that can be used to detect unsatisfiability in an ORM conceptual schema[3]. We adopt the ORM formalization and syntax as found in [H89,H01], except three things. First, although ORM supports n-ary predicates, only binary predicates are considered. Second, our approach does not support objectification, or the so-called nested fact-types in ORM. Finally, our approach does not support the derivation constraints that are not part of the ORM graphical notation.

Pattern 1 (Top common supertype)

In this pattern, subtypes that do not have a top common supertype are detected. In ORM, all object-types are assumed by definition to be mutually exclusive, except those that are subtypes of the same supertype. Thus, if a subtype has more than one supertype, these supertypes must share a top supertype; otherwise, the subtype cannot

[3] An extended version of this work can be found in [J05].

be satisfied. In Fig. 2 the object-type C cannot be satisfied because its supertypes A and B do not share a common supertype, i.e. A and B are mutually exclusive.

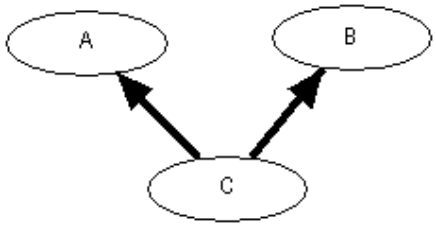


Fig. 2. Subtype without a top common supertype.

Formally, for each subtype T in the schema, let $\mathsf{T.DirectSupers}$ be the set of all n direct supertypes of T. Let $\mathsf{T.DirectSupers_i.Supers}$ be all supertypes of the i-th direct supertype of T. If $(\mathsf{T.DirectSupers_1.supers} \cap ... \cap \mathsf{T.DirectSupers_n.supers}) = \varnothing$, then the object-type T cannot be satisfied. See the appendix for the implementation detail.

Pattern 2 (Exclusive constraint between types)

In this pattern, subtypes of mutually exclusive supertypes (caused by an exclusive constraint) are detected. Fig. 3 shows a case where D cannot be satisfied because its supertypes are mutually exclusive. The set of instances of D is the intersection of the instances of B and C, which is an empty set according to the exclusive constraint between B and C.

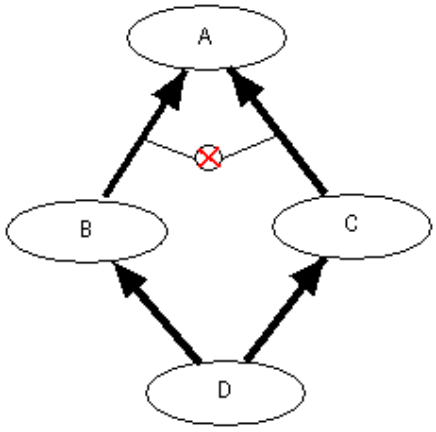


Fig. 3. Subtype with exclusive supertypes.

Formally, for each exclusive constraint between a set of object-types $\mathsf{T} = \{\mathsf{T_1}, ... \mathsf{T_n}\}$, let $\mathsf{T_i.Subs}$ be the set of all possible subtypes of the object-type T_i, and $\mathsf{T_j.Subs}$ be the set of all possible subtypes of the object-type T_j, where $i \neq j$, the set $(\mathsf{T_i.Subs} \cap \mathsf{T_j.Subs})$ must be empty. See the appendix for the implementation detail.

Pattern 3 (Exclusion-Mandatory)

In this pattern, contradictions between exclusion and mandatory constraints are detected. In Fig. 4, we show three examples of unsatisfiable schemes.

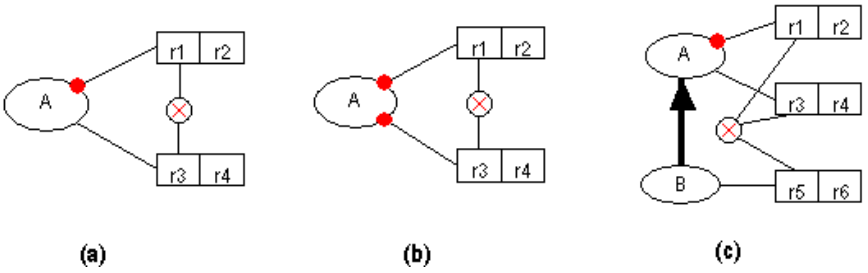


Fig. 4. Unsatisfiable schemes because of mandatory and exclusion conflicts.

In the first case (a), the role r3 will never be played. The mandatory and exclusion constraints restrict that each instance of A must play r1 and the instance that plays r1 cannot play r3. In the second case (b), both r1 and r3 will never be played. According to the two mandatory constraints, each instance of A must play both r1 and r3. At the same time, according to the exclusion constraints, an instance of A cannot play r1 and r3 together. Likewise, in the third case (c), r3 and r5 will never be played. As B is a subtype of A, instances of B inherit all roles and constraints from A. For example, if an instance of B plays r5, then this instance – which is also instance of A – cannot play r1 or r3. However, according to the mandatory constraint, each instance of A must play r1 and, according to the exclusion constrain, it cannot play r1, r3 and r5 all at the same time. In general, a contradiction occurs if an object-type plays a mandatory role that is exclusive with other roles played by this object-type or one of its subtypes. Formally, for each exclusion constraint between a set of single roles R, let $R_i.T$ be the object-type that plays the role R_i, $R_i \in R$. For each (R_i, R_j), where $i \neq j$ and R_i *is mandatory*, If $R_i.T = R_j.T$ or $R_j.T \in R_i.T.Subs$ -where $R_i.O.Subs$ is the set of all subtypes of the object-type $R_i.O$, then some roles in R cannot be populated.

Pattern 4 (Frequency-Value)

In this pattern, contradictions between value and frequency constraints are detected**.**

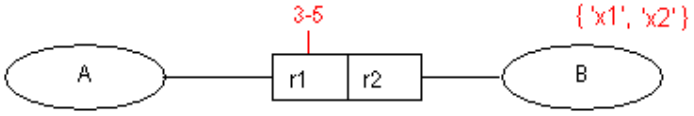


Fig 5. Contradiction between value and frequency constraints.

In Fig. 5, the role r1 cannot be populated. If the frequency constraint FC(3 - 5) on r1 is satisfied, each instance of A must play r1 at least three times, and thus three different instances of B are required. However, there are only two possible instances of B, which are declared by the value constraint {‘x1’, ‘x2’}. For each fact-type

($A\ r\ B$), let c be the number of the possible values of B that can be calculated from its value constraint, and let FC($n-m$) be a frequency constraint on the role r, then c must be equal or more than n. Otherwise, the role r cannot be satisfied, as the value and the frequency constraints contradict each other.

Pattern 5 (Value-Exclusion-Frequency)

In this pattern, contradictions between value, exclusion, and frequency constraints are detected. Fig. 6 shows a particular contradiction between those three constraints. Due to the frequency constraint, there should be at least two different values to populate r1. In order to populate r3, we need, by the exclusion constraint, a value different from the two for role r1. In total, we thus need three different values in order to be able to populate both r1 and r2, but this contradicts with the value constraint on object-type A: we only have 2 values at our disposal. Note that any combination with only two of the three constraints does not amount to unsatisfiability; we explicitly need the combination of the three of them.

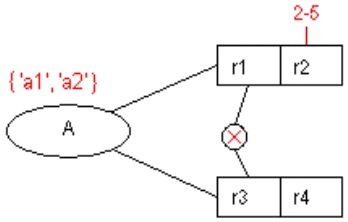


Fig. 6. Contradiction between value, exclusion, and frequency constraints.

A special case occurs in the absence of frequency constraints, e.g., Fig. 7: according to the exclusion constraint, there should be at least three different values of A to play r1, r2 and r3. However, according to the value constraint, there are only two possible values of A.

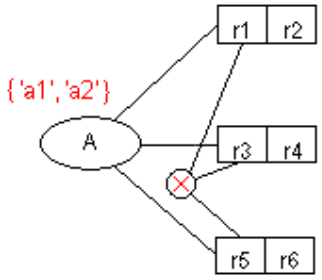


Fig. 7. Contradiction between value and exclusion constraints.

We assume that exclusion constraints are in their most compact form. For example, the exclusion constraint in Fig. 7 is the compact representation of three different exclusion constraints: between r1 and r3, r3 and r5, and r1 and r5.

Formally, for each exclusion constraint, let $R=\{R_1,\ldots,R_n\}$ be the set of roles participating in this constraint. With each of those roles R_i, we associate the inverse role S_i, and we let f_i be the minimum of the frequency constraint on S_i (if there is no

frequency constraint on S_i, we take f_i equal to 1). Let T be the object-type that plays all roles in R. Let C be the number of the possible values of T, according to the value constraint. C *must always be more than or equal to* $f_1+\ldots+f_n$. Otherwise, some roles in R cannot be satisfied. See the appendix for the implementation detail. Note that this pattern is actually a generalization of the previous pattern where there are no exclusion constraints. However, the current pattern explicitly focuses on the exclusion constraints attached to a role, taking into account the frequency constraints, to decide whether some roles are unsatisfiable. As pattern 4 does not contain exclusion constraints, a similar strategy would not work.

Pattern 6 (Set-comparison constraints)

In this pattern, contradictions between exclusion, subset, and equality constraints are detected. Fig. 8 shows a contradiction between the exclusion and the subset constraints. This contradiction implies that both predicates cannot be populated.

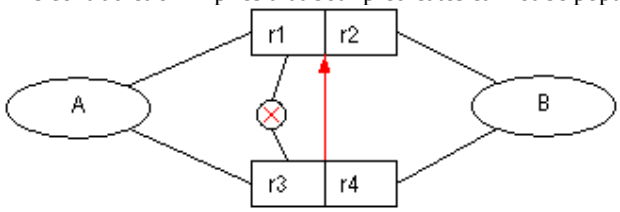


Fig. 8. A non fact-type populatable schema.

The exclusion constraint between the two roles r1 and r3 means that their populations should be distinct. However, in order to satisfy the subset constraint between (r1, r2) and (r3, r4), the populations of r1 and r3 should not be distinct. In other words, the exclusion constraint between r1 and r3 implies an exclusion constraint between (r1, r2) and (r3, r4) [H89], which contradicts any subset or equality constraint between both predicates. Fig. 9 shows the implications for each set-comparison constraint that might be declared between parts of role sequences. These implications are taken into account when reasoning for contradictions between the three set-comparison constraints.

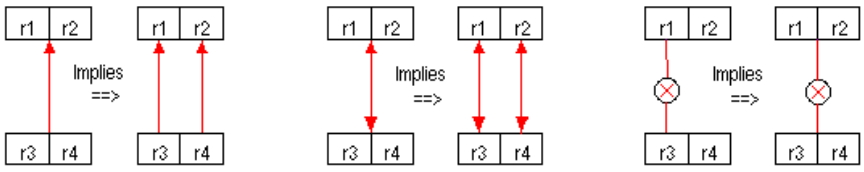


Fig. 9. Main set-comparison implications.

In addition, an equality constraint is equivalent to two subset constraints. Hence, we refer to a subset or an equality constraint as a SetPath. Formally, for each exclusion constraint between A and B: If A and B are two predicates, there should not be any (direct or implied) SetPath between these predicates; If A and B are single roles, there should not be any (direct or implied) SetPath between both roles or between the predicates that include these roles. Otherwise, the two predicates cannot be populated,

as the two constraints contradict each other. See the appendix for the implementation detail.

Pattern 7 (Uniqueness-Frequency)

In this pattern, all occurrences of a uniqueness constraint that contradicts with a frequency constraint on the role are detected. E.g., in Fig. 10 the uniqueness constraint indicates that the role r1 should be played by at most one element, while the frequency constraint demands that there are at least 2 and at most 5 participants in the role (denoted as FC(2-5)). It is thus impossible to populate r1. Formally, unsatisfiability of a role occurs if there is a frequency constraint FC(min-max) and a uniqueness constraint on some role (or predicate) r where min is strictly greater than 1. See the appendix for the implementation detail.

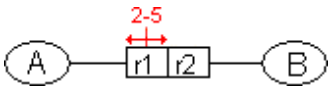


Fig. 10. Unsatisfiability of frequency and uniqueness constraint.

Pattern 8 (Ring constraints)

ORM allows ring constraints to be applied to a pair of roles that are connected directly to the same object-type in a fact-type, or indirectly via supertypes. Six kinds of ring constraints are supported by ORM: antisymmetric (ans), asymmetric (as), acyclic (ac), irreflexive (ir), intransitive (it), and symmetric (sym) [H01,H99]. For example, Fig. 11 shows an irreflexivity on the *Sister Of* role, indicating that no woman is her own sister.

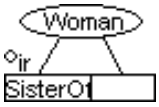


Fig. 11. Irreflexivity on the ring constraint

The relationships between the six ring constraints are formalized by [H01] using the Euler diagram as in Fig. 12. This formalization indeed helps to visualize the implication and incompatibility between the constraints. For example, one can see that acyclic implies reflexivity, intransitivity implies reflexivity, the combination between antisymmetric and irreflexivity is exactly asymmetric, and acyclic and symmetric are incompatible, i.e. their combination leads to unsatisfiability).

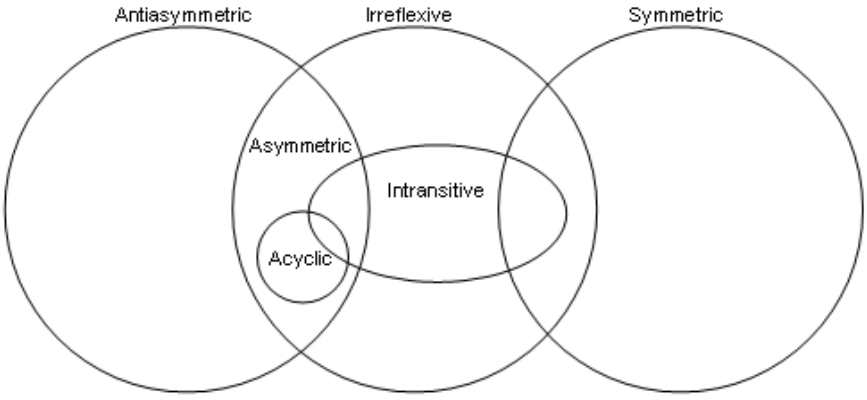


Fig. 12. Relationships between ring constraints [H01].

In general, unsatisfiability occurs if two ring constraints that are disjoint in the Euler diagram are used. Based on the Euler diagram, we derive in Table 1. all possible compatible combinations of the six ring constraints. Combinations that do not appear in the table are incompatible and lead to unsatisfiable roles, e.g., (Sym, it) and (Ans), (Sym, it) and (It, ac), or (Ans, it) and (Ir, sym).

No.	Constraint$_1$	Constraint$_2$	Combination
1	Ans	Ir	As
2	Ans	As	as
3	Ans	It	Ans, it
4	Ans	Ac	ac
5	Ir	Sym	Ir, sym
6	Ir	As	as
7	Ir	It	It
8	Ir	Ac	Ac
9	Sym	It	Sym, it
10	As	It	As, it
11	As	Ac	ac
12	it	ac	It, ac
13	Ans, it	Ans	Ans, it
14	Ans, it	Ir	Ans, it
15	Ans, it	As	Ans, it
16	Ans, it	It	Ans, it
17	Ans, it	Ac	it, ac
18	Ir, sym	Ir	Ir, sym
19	Ir, sym	Sym	Ir, sym

No.	Constraint$_1$	Constraint$_1$	Combination
20	Ir, sym	It	It, sym
21	Sym, it	Ir	Sym, it
22	Sym, it	Sym	Sym, it
23	Sym, it	It	Sym, it
24	As, it	Ans	As, it
25	As, it	Ir	As, it
26	As, it	As	As, it
27	As, it	It	As, it
28	As, it	Ac	Ac, it
29	It, ac	Ans	It, ac
30	It, ac	Ir	It, ac
31	It, ac	As	It, ac
32	It, ac	It	It, ac
33	It, ac	Ac	It, ac
34	Ans, it	As, it	it, as
35	Ans, it	It, ac	it, ac
36	Ir, sym	Sym, it	sym, it
37	As, it	It, ac	it, ac

Table. 1. All possible compatible combinations or ring constraints.

Pattern 9 (Loops in Subtypes)

In this pattern, loops in the subtype relation are detected. Since in ORM, the population of a subtype is a strict subset, i.e. a subset and not equal, of the population of its supertype [H01], one cannot have loops. Otherwise, one would have that a population is a strict subset of itself, which is not possible. In Fig. 13, none of the object-types A, B, or C can be satisfied since they form a loop.

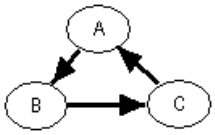


Fig. 13. Loop in subtypes.

Formally, for each subtype T in the schema, let T.Supers be the set of all supertypes of T . If T in T.Supers, then the object-type T cannot be satisfied. See the appendix for the implementation detail. Note that there is no analogous pattern for subset constraints; no strict subset relation is required for subset constraints, such that loops in subset constraints imply equality of the involved roles but do not lead to unsatisfiability in general.

3 Related Work

In [H89], 7 formation rules for constraints on ORM conceptual schemes are described. We discuss to which extent these rules can also be used for detecting unsatisfiability of roles and how they relate to the patterns described in the previous section.

Formation rule 1 *(A frequency constraint of 1 is never used (the uniqueness constraint must be used instead))* and rule 2 *(A frequency constraint cannot span a whole predicate)* prefer one syntactical form over another (rule 1) or prohibit a, from a logical perspective, nonsensical[4] frequency constraint (rule 2). Rule 1 is, however, not relevant, where we call a rule *relevant* if it is relevant from an unsatisfiability detection perspective, i.e. a rule is relevant, if in case it is violated, there is an unsatisfiable role. Regarding rule 2, as the population of an ORM predicate is basically a set, any frequency constraint FC(min-max) where min is strictly greater then 1 leads to unsatisfiability. Rule 2 is, however, too strict in the sense that a frequency constraint FC(1-max), although redundant, does not lead to unsatisfiability. Pattern 7 takes care of the latter case (where it is assumed, as is implicit in ORM, that a predicate is spanned by a uniqueness constraint). Note that we are only interested in unsatisfiability; from a modeling perspective the formation rules are most certainly useful, in the sense that they, e.g., avoid adding redundant constraints to the schema.

Rule 3 *(No role sequence exactly spanned by a uniqueness constraint can have a frequency constraint)* is again too strict by itself to be relevant for unsatisfiability. For example, a constraint FC(1 – 5) and a uniqueness constraint on the same role do not yield an unsatisfiable role. They are, however, equivalent with FC(1 – 1) or with a mandatory plus a uniqueness constraint, thus, from a modeling perspective, formation rule 3 makes sense, but it does not necessarily lead to an unsatisfiable role. We loosened up rule 3 to take this into account in pattern 7.

Rule 4 *(No uniqueness constraint can be spanned by a longer uniqueness constraint)* is again not relevant for unsatisfiability. Rule 5 *(An exclusion constraint cannot be*

[4] Nonsensical, since predicates are interpreted as sets, where each element in a set is, by definition of sets, unique in that set.

specified between roles if at least one of these roles is marked as mandatory) is exactly pattern 3; we made it explicit that the rule applies to subtypes as well.

Rule 6 *(An exclusion constraint cannot be specified between two roles attached to object-types one of which is specified as a subtype of the other)* is not relevant for unsatisfiability. There are ORM conceptual schemes violating rule 6, although all roles are satisfiable, e.g., Fig. 14.

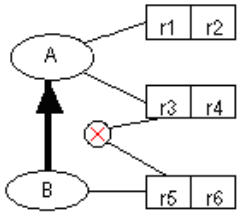


Fig. 14. ORM schema violating formation rule 6 but with satisfiable roles.

For example, populate r5 with some 'a', then 'a', by the subtyping, must play one of the roles r1 or r3. It cannot play r3 due to the exclusion constraint but nothing keeps it from playing r1.

The last formation rule, rule 7 *(A frequency constraint with upper bound n cannot be specified on a role sequence if n is less than the product of the maximum cardinalities of the other role populations for the predicate)*, is covered by pattern 4 since we restrict ourselves to binary predicates[5]. The 7 ORM formation rules thus provide useful criteria for constructing ORM schemes: in a lot of cases they avoid unsatisfiability as well as implied (redundant) constraints. However, the rules mix both syntactical and semantical criteria, occasionally yielding too strict criteria for detecting unsatisfiability. The constraint patterns, described in Sec. 3, focus on the semantical aspect of unsatisfiability only.

In [DMV], the RIDL-A module of the RIDL* workbench, a database engineering tool based on the NIAM methodology [VB82], checks whether a conceptual schema is correctly constructed. Since NIAM is the predecessor of ORM, it is interesting to compare the criteria RIDL-A employs to our patterns. Of particular interest in the RIDL-A module are the Validity Analysis (rules V1-V6) and Set Constraint Analysis (rules S1 – S4) parts.

It appears that none of the Validity Analysis rules are relevant for unsatisfiability. The Set Constraint Analysis part contains 4 rules dealing with three types of constraints: subset, equality, and exclusion. S1 and S3 say that a subset (resp. equality) constraint may not be superfluous[6]. Although interesting from a modeling perspective, neither S1 nor S3 lead to unsatisfiability of roles in itself. S2 *(A subset constraint may not contain any loops)* is not relevant for unsatisfiability; the population of the roles would be equal but can be non-empty. Note that we use the definition of subset constraints on predicates as in [H89], i.e. a role r1 is a subset of r2 if every element

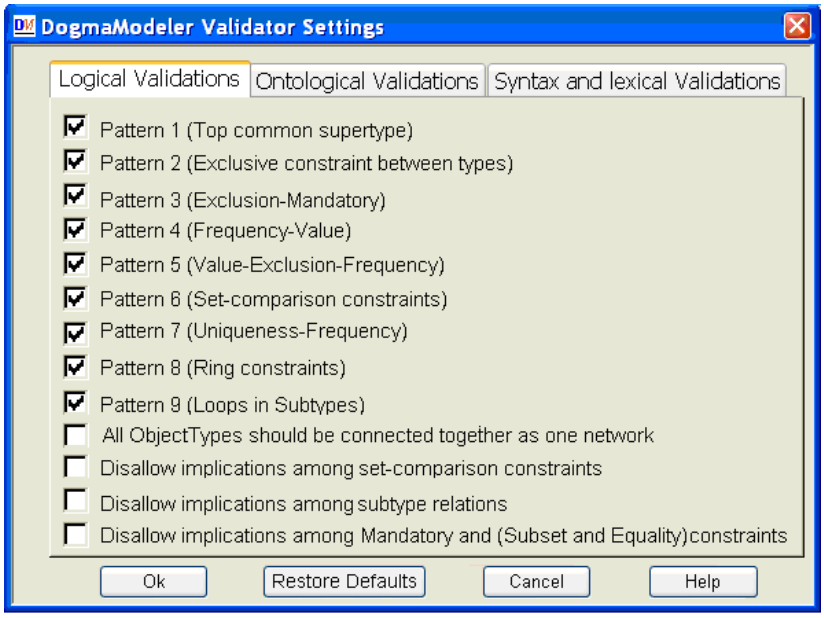
[5] Maximal cardinalities in [H89] correspond to value constraints.
[6] A constraint is superfluous – or implied – if it can be derived from other constraints.

playing role r1 also plays r2. In particular, r1 does not need to be a *strict* subset of r2: they may be equal. S2 is relevant for subset constraints between subtypes since those are strict; we covered this with pattern 9.

Finally, S4 *(The OTSETS[7] involved on an exclusion constraint may not have a common subset)* is a valid condition for detecting inconsistency. It is, however, too general, in the sense that it is actually the definition of an exclusion constraint, and does not indicate how the exclusion might yield unsatisfiable roles.

4 Implementation and Discussion

The DogmaModeler, an ontology and business rules engineering tool, implements the patterns described in this paper. Fig. 15 displays these patterns as a menu in the DogmaModeler Validator Settings window[8]. Users can choose to enable or disable the enforcement of these validation patterns when reasoning about the satisfiability of an ORM model. The DogmaModeler typically implements the satisfiability patterns that we have developed in Sec. 2. In the appendices, we present 9 algorithms, which are written in a JAVA-like code, to implement the 9 patterns in DogmaModeler. One can see, from these algorithms, that DogmaModeler does not only detect unsatisfiable ORM models, but also, it gives (through the generated message) details about the detected problems, such as, which constraints cause the unsatisfiability, the problems with the other constraints, etc.



[7] The OTSET of a role corresponds, roughly, to the population of a role.
[8] The specification of the last three implication patterns is adopted from [H89].

Fig. 15. DogmaModeler's support of logical validations.

Experience in developing ontologies in DogmaModeler shows that detecting unsatisfiability in an interactive manner helps ontology builders in quick detection of mistakes. In other words, we found that interactive detection of unsatisfiability improves the modeling skills of ontology builders, especially those who are not well trained in ontology modeling and logics.

It is probably worth to note that our motivation for developing this patterns-approach was derived from our experience in using ORM to develop a Customer Complaint Ontology [J05,JVM03], which was done within the CCFORM project, IST-2001-38248. This ontology was built by 10s of lawyers, and the presented reasoning patters were applied to guide those lawyers to detect inconsistency problems in early phases. One of the interesting lessons we have learned in this project is that the implementation of the patterns (*in an interactive manner* during the ontology modeling process) enabled the lawyers to learn how to avoid such mistakes the next time. Some of them even admitted that they understood some logics from their experience in using DogmaModeler[9].

As we have mentioned earlier, we have also tackled the ORM satisfiability problem in another way. We have mapped ORM into the ***DLR*** Description Logic [JF05], DLR is a powerful and decidable fragment of first order logic [CDLNR98]. This mapping[10] provides us a complete reasoning support for ORM schemes, i.e. users are able to check (strong and weak) satisfiability of an ORM schema by satisfiability checking of the corresponding ***DLR*** knowledge base, which can be done using RACER[11].

On comparison between pattern detection approaches and a complete reasoning procedure in description logic, we find that both approaches complement each other. Pattern detection approaches are easy to implement, specially in interactive modeling tools, and is bound to be cheaper in terms of reasoning time than a complete reasoning procedure. A complete procedure, e.g. by mapping an ORM schema to a DL knowledge base and deploying existing DL reasoners, typically is exponential. So even in the presence of a complete reasoner, the patterns can be used to quickly detect any "trivial" inconsistencies before calling the more expensive (but complete) procedure, thus speeding up the modeling process. Last but not least, a pattern detection algorithm can be designed and optimized for certain usages, e.g. specifically for ORM unsatisfiability, while a DL translation would rely on a general knowledge base reasoning which e.g. cannot be optimized for ORM constructs.

---

[9] More details about DogmaModeler and our experience in CCFORM are not presented in this paper because of the space limitations, but can be found e.g. in [J05], [J06], [JVM03] and [JDM03].

[10] Some ORM constrains (but are rarely used in practice) cannot be mapped into DLR such as ring constraints, frequency constraints on several roles, etc.

[11] http://www.racer-systems.com/products/download/index.phtml (July 2005).

5 Conclusions and Further Research

We presented 9 patterns to detect unsatisfiability of roles in an ORM schema, and discussed the relation with existing rules in the literature. The implementation and application of this approach has been illustrated and discussed.

In the future, we intend to devise more patterns for unsatisfiability checking, e.g., checking which combinations that involve more than 2 constraints lead to unsatisfiability while leaving out one constraint would not lead to unsatisfiability (as in pattern 5). Moreover, we intend to extend our approach to detect unsatisfiability for ORM derivation rules, assertional knowledge, etc.

Although the 9 patterns are patters that arise frequently in faulty modeling, they are by no means complete. E.g., one could demand that for irreflexive roles at least 2 different values need to be present.

One may notice that our patterns can be easily translated to other knowledge representation languages, especially for ontology and business-rules modeling tools[12].

References

[BCMNP03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. The Description Logic Handbook. *Cambridge University Press*, 2003.

[BGH99] Bird, L., Goodchild, A., Halpin, T.A.: Object Role Modelling and XML-Schema. In: Laender, A., Liddle, S., Storey, V. (eds.): *Proc. of the 19th International Conference on Conceptual Modeling (ER'00)*. LNCS, Springer, 1999.

[BHW91] P. van Bommel, A.H.M. ter Hofstede, and Th.P. van der Weide. Semantics and verification of object role models. *Information Systems*, 16(5), pp. 471-495, 1991.

[BvHHHMP04] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. `http://www.w3.org/TR/owl-ref/`, 2004.

[CDLNR98] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, R. Rosati. Information integration : Conceptual Modeling and reasoning support. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS'98)*, pp. 280-291, 1998.

[DJM02a]: Demey, J., Jarrar, M., Meersman, R.: A Conceptual Markup Language that supports interoperability between Business Rule modeling systems. *Proc. of the Tenth International Conference on Cooperative Information Systems (CoopIS 02)*. Springer LNCS 2519, pp. 19-35, 2002.

[DMV] O. De Troyer, R. Meersman, and P. Verlinden. RIDL* on the CRIS case: A Workbench for NIAM. *Technical report*. INFOLAB, Tilburg University, The Netherlands.

[H] T. Halpin. Object-Role Modeling: an overview. White paper, *http://www.orm.net.*

[H01] T. Halpin. Information Modeling and Relational Databases. 3rd edn. Morgan-Kaufmann, 2001.

[H89] T. Halpin. A logical analysis of information systems: static aspects of the data-oriented perspective. *PhD thesis*, University of Queensland, Brisbane, Australia, 1989.

[12] One of our master students aims to implement these patterns in Protégée, as part of his thesis on interactive ontology modeling.

[H97] Halpin, T.: An Interview- Modeling for Data and Business Rules. *In: Ross, R. (eds.): Database Newsletter*. vol. 25, no. 5. (Sep/Oct 1997). -This newsletter has since been renamed Business Rules Journal and is published by Business Rules Solutions, Inc.

[J06]: Jarrar, M.: Towards the notion of gloss, and the adoption of linguistic resources in formal ontology engineering. Proceeding of the 15th International World Wide Web Conference, WWW2006. Edinburgh, Scotland. May 2006. ACM, 2006.

[J05] M. Jarrar. Towards Methodological Principles for Ontology Engineering. *PhD thesis*, Vrije Universiteit Brussel, 2005.

[JF05] Jarrar, M., Franconi, E.: Mapping ORM into the DLR description logic. Technical Report, August 2005.

[JVM03] Jarrar, M., Verlinden, R., Meersman, R.: Ontology-based Customer Complaint Management. In: Jarrar M., Salaun A., (eds.): Proceedings of the workshop on regulatory ontologies and the modeling of complaint regulations, Catania, Sicily, Italy. Springer Verlag LNCS. Vol. 2889. November (2003) pp. 594–606

[N99] North, K.: Modeling, Data Semantics, and Natural Language. *In: New Architect magazine*, 1999.

[T96] O. De Troyer. A formalization of the binary Object-role Model based on Logic. *Data & Knowledge Engineering* 19, North-Holland, Elsevier, pp. 1-37, 1996.

[TM95] O. De Troyer and R. Meersman. A Logic Framework for a Semantics of Object-Oriented Data Modelling. In *Proc. Of 14th International Conference Object-Orientation and Entity-Relationship Modelling (OO-ER'95)*, LNCS 1021, pp. 238-249, Springer, 1995.

[VB82] G. Verheijen and P. van Bekkum. NIAM, aN Information Analysis Method. In Proc. Of the *IFIP Conference on Comparative Review of Information Systems Methodologies*, North-Holland, 537-590, 1982

Appendix: Algorithms

Pattern 1

```
For each subtype T[x] {
 Let T[x].DirectSupers = the set of all direct supertypes of T[x].
 n = T[x].DirectSupers.size
 If ( n > 1)   {
   For (i = 1 to i=n)     {
     Let T[x].DirectSupers[i].Supers = the set of all possible supertypes
                                              of T[x].DirectSupers[i]    }
   // if the intersection of all T[x].DirectSupers[i].supers is not empty,
      then the composition is not satisfiable.
   if (Intersection(T[x].DirectSupers[1].supers, … T[x].DirectSupers[n].supers))
       is empty  {
        Satisfiability = false
        Message= ("The subtype T[x].DirectSupers[i] cannot
             be satisfied as its supertypes do not have a top common supertype.")        }} }
```

Pattern 2

```
For each exclusive constraint Exv[x] {
  Let Exv[x].T = the set of the object-types participating in Exv[x].
  //For each pair of object-types participating in the exclusion constraint:
  For (i = 1 to i = Exv[x].T.size) {
     For (j = 1 to j = Exv[x].T.size) {
        If (i not equal j) {
           Let Exv[x].T[i].Subs = the set of subtypes of the object-type Exv[x].T[i].
           Let Exv[x].T[j].Subs = the set of subtypes of the object-type Exv[x].T[j].
           S = IntersectionOf(Exv[x].T[i].Subs, Exv[x].T[j].Subs)
           If (S is not empty) {
              Satisfiability = false
```

```
            Message = (“all subtypes in <S> cannot be instantiated because of <Exv[x]>“) }}}}   }
```

Pattern 3

```
For each exclusion constraint Exs[x] between a set of single roles {
  Let Exs[x].roles = the set of all roles participating in Exs[x].
  For (i=1 to Exs[x].roles.size)
    If (Exs[x].roles[i].Mandatory = true) {
      For (j=1 to Exs[x].roles.size) {
        If (I not equal j){
        Let Exs[x].roles[i].T = the object-type that plays the role Exs[x].roles[i]
        Let Exs[x].roles[j].T = the object-type that plays the role Exs[x].roles[j]
        Let Exs[x].roles[i].T.Subs = the set of all subtypes of Exs[x].roles[i].T
        If (Exs[x].roles[i].T = Exs[x].roles[j].T) OR
                                    In(Exs[x].roles[j].T, Exs[x].roles[i].T.Subs ) {
            Satisfiability = false
            Message = (“There are some roles in <Exs[x].roles> that cannot
              be instantiated because of the <Exv[x]>“)}}}}}
```

An alternative but more compact algorithm can be:

```
For each exclusion constraint Exs[x] between a set of single roles {
  Let Exs[x].roles = the set of all roles participating in Exs[x].
  Let MandRoles = the set of all mandatory roles from Exs[x].roles.
  If (MandRoles  is not empty)
     For (i=1 to ManRoles.size)
       For (j=1 to Exs[x].roles.size)
        Let MandRoles[i].T = the object-type that plays the role MandRoles[i]
        Let Exs[x].roles[j].T = the object-type that plays the role Exs[x].roles[j]
        Let Exs[x].roles[j].T.Subs = the set of all subtypes of Exs[x].roles[j].T
        If Not In(MandRoles[i].T, Exs[x].roles[j].T.Subs)
            Satisfiability = false
            Message= (“There are some roles in
              <Exs[x].roles> that cannot be populated because of the <Exv[x]>“)}}}}   }
```

Pattern 4

```
For each frequency constraint F[x] {
    Let F[x].min = the lower bound of the frequency constraint F[x].
    Let T = the object-type that is played by the role holding F[x].
    Let T.Values = the value constraint on T.
    // if there is no value constraint on T, then T.Values = null
    If (T.Values is not null) and (T.Values.size < F[x].min) {
        Satisfiability = false.
        Message =(“the role <T.r> cannot be instantiated because the
            <F[x]> and the <T.Values> contradict each other”). } }
```

Pattern 5

```
For each exclusion constraint Exs[x] between a set of single roles {
  Let Exs[x].Roles = the set of roles participating in the exclusion Exs[x].
  Let Exs[x].InvRoles = the set of inverse roles of Exs[x].Roles.
  For ( Si in  Exs[x].InvRoles) {
              If ( there is frequency constraint on Si )
                          fi = minimum freq. const. on Si;
              else fi = 1;   }
  Let F = sum(fi).
  Let O = the object-type that plays all roles in Exs[x].Roles.
  Let O.Values = the value constraint on O.
  // if there is no value constraint on O, then O.Values = null
  If (O.Values is not null) and (O.Values.size < F) {
        Satisfiability = false.
       Message =(“Some roles in <Exs[x].Roles> cannot be instantiated because
            the <Exs[x]> and the <O.Values> contradict each other”).} }
```

Pattern 6

```
For each exclusion constraint Exs[x] {
 If (Exs[x] between predicates) {
    Let Exs[x].predicates = the set of all predicates participating in Exs[x].
    \\ For each pair of predicates participating in the exclusion
    For (i = 1 to i = Exs[x].predicates.size) {
      For (j = 1 to j = Exs[x].predicates.size) {
        If (i not equal j) {
          Sp = GetSetPathsBetween(Exs[x].Predicates[i], Exs[x].Predicates[j])
          // Sp is the set of all subset or equality constraints that specify or imply a
          // SetPath between the current tuple of predicates.
          If (Sp is not empty) {
            Satisfiability = false.
            Message = ("the exclusion constraint <Exs[x]> contradicts some subset
                        and/or equality constraints on the predicates in <Sp>").}}}}}
 Else { // then the Exs[x] is between roles
    Let Exs[x].roles = the set of all roles that participate in Exs[x].
    \\ For each pair of roles participating in the exclusion constraint
    For (i = 1 to i = Exs[x].roles.size) {
      For (j = 1 to j = Exs[x].roles.size) {
        If (i not equal j) {
          Sr = GetSetPathsBetween(Exs[x].roles[i], Exs[x].roles[j])
          // Sr is the set of all subset or equality constraints that specify or imply a
          // SetPath between the current tuple of roles.
          Sp = GetSetPathsBetween(Exs[x].Predicates[i], Exs[x].Predicates[j])
          // Sp is the set of all subset or equality constraints that specify or imply a
          // SetPath between the predicates of the current tuple of roles.
          If (Sr is not empty) OR (Sp is not empty) {
            Satisfiability = false.
            Message = ("the exclusion constraint <Exs[x]> contradicts some Subset
                 and/or equality constraints on the predicates in Sp"). }}}}}}}
```

Pattern 7

```
For each frequency constraint F[x] {
    Let F[x].min = the lower bound of the frequency constraint F[x].
   Let R = the role (predicate) on which F[x] is placed.
    If ( there is uniqueness constraint on R ) and ( F[x].min > 1 )
       Satisfiability = false
        Message= ("The frequency constraint F[x] cannot  be satisfied as it conflicts with a
uniqueness constraint.") }
```

Pattern 8

```
For each role R {
  Let RC = the set of ring constraints on R
   If ( RC not allowed according to Table 1.)
        Satisfiability = false
        Message= ("The ring constraints RC cannot  be satisfied.") }
```

Pattern 9

```
For each subtype T[x] {
 Let T[x].Supers = the set of all supertypes of T[x].
If ( T[x] in T[x].Supers )   {
        Satisfiability = false
        Message= ("The subtype T[x] is part of a loop, thus it cannot be satisfied.")       }}}
```