International Journal on Artificial Intelligence Tools

(Volume 17, Number 1)

Towards Pattern-based Reasoning for Friendly Ontology Debugging

Mustafa Jarrar
HPCLab, University of Cyprus
STARLab, Vrije Universiteit Brussel
mjarrar@cs.ucy.ac.cy

Stijn Heymans
DERI, University of Innsbruck, Austria
stijn.heymans@deri.org

Reasoning with ontologies is a challenging task specially for non-logic experts. When checking whether an ontology contains rules that contradict each other, current description logic reasoners can only provide a list of the unsatisfiable concepts. Figuring out why these concepts are unsatisfiable, which rules cause conflicts, and how to resolve these conflicts, is all left to the ontology modeler himself. The problem becomes even more challenging in case of large or medium size ontologies, because an unsatisfiable concept may cause many of its neighboring concepts to be unsatisfiable.

The goal of this article is to empower ontology engineering with a user-friendly reasoning mechanism. We propose a pattern-based reasoning approach, which offers 9 patterns of constraint contradictions that lead to unsatisfiability in Object-role (ORM) models. The novelty of this approach is not merely that constraint contradictions are detected, but mainly that it provides the causes and suggestions to resolve contradictions. The approach is implemented in the DogmaModeler ontology engineering tool, and tested in building the CCFORM ontology. We discuss that, although this pattern-based reasoning covers most of contradictions in practice, compared with description logic based reasoning, it is not complete. We argue and illustrate both approaches, pattern-based and description logic-based, their implementation in the DogmaModeler, and conclude that both complement each other from a methodological perspective.

Keywords: Ontology Engineering; Reasoning; Satisfiability; Model Verification; Debugging; Ontology Tools; Object Role Modeling; ORM;

1. Introduction and Motivation

In many domains the ontology building process is difficult and time consuming. Typically, this is because it is difficult for domain experts to understand and use ontology languages. Current ontology languages (and tools) require an understanding of their underpinning logic. The limitation of these types of ontology languages and tools is not that they lack expressiveness or logical foundations. Instead, it is in their capability to be used by subject matter experts.

Furthermore, reasoning on ontologies is a difficult task not only for non-logic experts, but even for ontology engineers themselves. For example, when checking whether an ontology contains rules that contradict each other, *current description logic reasoners can only provide a list of the unsatisfiable concepts*[?,?,?,?]. Figuring out why these concepts are unsatisfiable[a], which rules cause conflicts, and how to resolve these conflicts, are all left to

[a]We say that a concept is unsatisfiable in case this concept cannot be instantiated, e.g. because of some contradic-

the ontology engineers themselves. The problem becomes even more challenging in case of large or medium size ontologies, because an unsatisfiable concept may cause many of its neighboring concepts to be unsatisfiable.

In previous research[?,?,?,?,?], we have proposed the use of the ORM conceptual modeling method as a graphical notation for ontology modeling. As we shall explain in the next section, the graphical expressiveness, the well-defined semantics, and the methodological and verbalization capabilities of ORM make it a good candidate as a graphical notation for modeling and representing ontologies. With this, non-IT trained industrial experts will be able to build axiomatized theories (such as ontologies, business rules, etc.) in a graphical manner, without having to know the underpinning logic or foundations.

The goal of this article is to empower ORM with a user-friendly reasoning mechanism[b]. We propose a pattern-based reasoning approach, which offers 9 patterns of constraint contradictions that lead to unsatisfiability. The novelty of this approach is not merely to detect constraint contradictions, but also to provide a *clear explanation* about: the detected contradictions, the causes, and suggestions on how to resolve these contradictions. In other words, our approach is motivated by the requirement that reasoning should be very friendly for non-logic experts, and should be easily implemented in *interactive modeling*.

This approach is implemented in the DogmaModeler ontology engineering tool, and tested in building the CCFORM ontology (a medium size legal ontology about customer complaints), which has been built by many lawyers. We shall illustrate (in section 4) how DogmaModeler guides ontology modelers to quickly detect unsatisfiability in early phases and does not require these modelers to have any background knowledge about logic or reasoning. One of the interesting lessons we have learned in the CCFORM project is that lawyers were able to intuitively understand their modeling mistakes and how to avoid it for next time. Some of them even admitted that they understood some logics from their experience in using DogmaModeler.

As we shall discuss in section 4.1, although our approach covers the most common unsatisfiability cases in practice, it cannot be complete. In other words, from a theoretical viewpoint there is no absolute guarantee that by passing all of the 9 patterns it means that the schema is strongly satisfiable. However, this is not the goal of this article. For complete reasoning, DogmaModeler also support description logic based reasoning using Racer, which acts as a background reasoning engine. We shall come back to this issue in section 4 and argue that the two (pattern-based and DL-based) approaches complement each other from a methodological viewpoint.

The need to trace the causes of contradictions is realized by several researchers in the context of debugging OWL ontologies[?,?], and in the context of ontology evolution[?]. Some researchers suggested to modify the internals and the tableau algorithms of the reasoning engines, which is called *glass box*. Other researchers suggested not to add this overhead to the reasoners internals, but a *black box* approach that analyze the reasoner's responses to allocate the sources of unsatisfiability. However, none of these approaches yields a user-friendly reasoning, and none is proven to be complete as they are mainly based on heuristics or certain types of conflicts. We shall comeback to this issue in section 6.

The remainder of the paper is organized as follows. In section 2 we give a background introduction about ORM, its formal semantics, and the types of satisfiability. In section 3, we introduce the 9 patterns of constraint contradictions that lead to unsatisfiability of ORM models. Section 4 presents the implementation of the patterns in DogmaModeler. Our experience and the lessons we learned during the CCFORM project are presented in

tions in the ontology. We shall explain unsatisfiability in the next sections.

[b] The approach presented in this paper can be fully applied for other ontology languages, such as OWL.

Section 4.1. Section 5 discusses related work from the DL and the ORM communities. Finally, section 6 presents our conclusions and directions for future work.

2. Object-Role Modeling (ORM)

ORM is a conceptual modeling method that allows a semantics of a universe of discourse to be modeled at a highly conceptual level and in a graphical manner. ORM[c] has been used commercially for more than 30 years as a database modeling methodology, and has recently becoming popular not only for ontology engineering but also as a graphical notation in other areas such as the modeling of business rules[?,?,?,?], XML-Schemes[?], data warehouses[?], requirements engineering[?,?], web forms[?,?], web engineering[?], etc.

ORM has an expressive and stable graphical notation. It supports not only n-ary relations and reification, but also a fairly comprehensive treatment of many "practical" and "standard" business rules and constraint types. These include identity, mandatoriness, uniqueness, subsumption, totality, exclusivity, subset, equality, exclusion, value, frequency, symmetry, intransitivity, acyclicity, derivation rules, and several others. Furthermore, compared with, for example, EER or UML, ORM's graphical notation is more stable since it is attribute-free; in other words, object types and value types are both treated as concepts. This makes ORM immune to changes that cause attributes to be remodeled as object types or relationships. Figure 1 shows an example of an ORM diagram. Concepts (also called

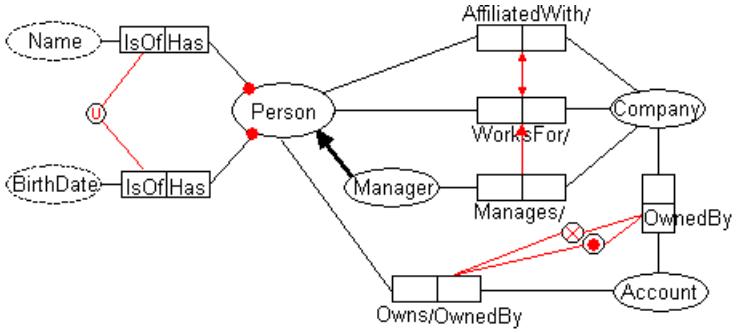

Fig. 1. Example of an ORM Schema.

object-types) are represented as ellipses, and *relations* are represented as rectangles. Each relation in ORM consists of one or more *roles*. For example, the relationship between the concepts Person and Account consists of the two co-roles, Owns and OwnedBy[d]. In description logic, this relation can be formalized as ($Person \sqsubseteq \forall Owns.Account$, $Account \sqsubseteq \forall OwnedBy.Person$, $OwnedBy \sqsubseteq Owns^{-}$). The thick arrow between Manager and Person denotes a *subsumption* relation ($Manager \sqsubseteq Person$). The dot (•) on the line connecting Person and Name represents a *mandatory* constraint ($Person \sqsubseteq \exists Has.Name$). The (U) represents an *external uniqueness*, meaning that a person can be uniquely denoted by the combination of his Name and Birthdate. The arrow between the relationships WorksFor/

[c]Many commercial and academic tools that support ORM solutions are available, including the ORM solution within Microsoft's Visio for Enterprise Architects[?], VisioModeler[?], NORMA[?],CaseTalk[?], Infagon[?], and DogmaModeler[?]. DogmaModeler and its support for reasoning will be presented in section 4

[d]Notice that the notion of role in ORM means an argument/component of a relationship. This is unlike e.g. description logics, where the notion of role means a binary relation. Furthermore, relationships in ORM do not have names, but at least one of its roles should have a linguistic label.

and Manages/ represents a *subset* constraint ($Manages \sqsubseteq WorksFor$), which means that if a person manages a company then this person must be employed by that company. The double-headed arrow between WorksFor/ and AffiliatedWith/ represents an *equality* constraint ($WorksFor \equiv AffiliatedWith$), which means that each person who works for a company is also affiliated with that company and vice versa. The ($\otimes$) between the two OwnedBy roles is an *exclusion* constraint that means: an account cannot be owned by a company and a person at the same time ($OwnedBy.Person \sqsubseteq \neg OwnedBy.Account$). The dot ($\odot$) on the two OwnedBy roles is called a *disjunctive mandatory*, it means each account must be owned by at least a company or a person ($Account \sqsubseteq \exists OwnedBy.Person \sqcup \exists OwnedBy.Person$).

ORM diagrams can be automatically verbalized into pseudo natural language sentences, i.e., all rules in a given ORM diagram can be translated into fixed-syntax sentences. For example, the mandatory constraint in Figure 1 is verbalized as: "Each *Person Has* at least one *Name*". The subset constraint is verbalized as: "If a *Person Manages* a *Company* then this *Person WorksFor* that *Company*", etc. Additional explanation can be found in [?] and [?] which provide sophisticated and multilingual verbalization templates. From a methodological viewpoint, this verbalization capability simplifies the communication with non-IT domain experts and allows them to better understand, validate, or build ORM diagrams. It is worthwhile to note that ORM is the historical successor of NIAM (Natural Language Information Analysis Method), which was explicitly designed (in the early 70's) to play the role of a stepwise methodology, that is, to arrive at the "semantics" of a business application's data based on natural language communication.

ORM Formal Semantics. ORM's formal specification and semantics are well-defined[?,?,?,?,?]. The most comprehensive formalization in first-order logic (FOL) was carried out by Halpin[?] in 1989. Later on, some specific portions of this formalization were re-examined, such as subtypes[?], uniqueness[?], objectification[?], and ring constraints [?].

Since reasoning on first-order logic is undecidable[?], the above formalizations do not enable automated reasoning on ORM diagrams, which comprises services such as detection of constraint contradictions (i.e. satisfiability), constraint implications, or inference.

ORM Satisfiability. Satisfiability checking (to detect constraint contradictions) is an important service in ontology modeling. 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 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, symptoms of a faulty model: there are too many constraints or constraints are too harsh[e].

To illustrate such contradictions, consider Figure 2, 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 Figure 2 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.

[e]We assume the universe of discourse (UoD) itself is consistent, such that faults in the model have their origin in the modeling and not in the UoD.

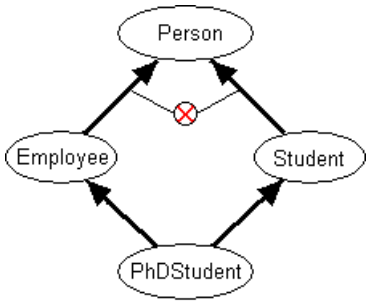

Fig. 2. Unsatisfiability of ORM Schema.

Types of Satisfiability. Formally, there are three types of satisfiability of an ORM schema[?]. 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 Figure 2. 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[f]. 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*. Notice that role satisfiability implies concept satisfiability only when all concepts are connected to roles.

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.

3. Unsatisfiability Patterns in ORM Conceptual Schemes

This section presents the 9 patterns of constraint contradictions that lead to unsatisfiability in an ORM conceptual schema[g]. Each pattern is explained by example, formal definition, and a Java-like algorithm that generates a message explaining the detected contradiction, its causes, and suggestions to resolve the contradiction. We adopt the ORM formalization and syntax as found in [?,?], 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.

[f] Figure 5 shows an ORM schema where all concepts are satisfiable (and thus the schema as a whole is satisfiable) but it fails because of role unsatisfiability (because role r3 cannot be satisfied).

[g] The first version of this work appeared in [?]. The second version[?] was presented at the IFIP.26 conference 2006, and at the Belgian-Dutch Database Day 2006. This version extends the previous versions in many directions, and benefits from the discussions and encouragements we received from many colleagues.

3.1. *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 be satisfied. In Figure 3 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.

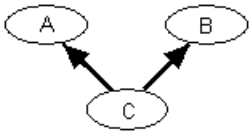

Fig. 3. Subtype without a Top Common Supertype.

Definition 1 Given a subtype T, let $\{DirectSuper_1, ..., DirectSuper_n\}$ be the set of all and only the direct supertypes of T. Let $Supers(DirectSuper_i)$ be the set of all possible supertypes of the i-th direct supertype of T, where $1 \leq i \leq n$. If $Supers(DirectSuper_1) \cap \ldots \cap Supers(DirectSuper_n) = \{\}$, then the object-type T cannot be satisfied.

Algorithm-1: For each subtype T[x] {
Let T[x].DirectSupers = The set of all and only the direct supertypes of T[x]
n = T[x].DirectSupers.size
If ($n \succ 1$){
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=(Contradiction!! Nothing can be an instance of $T[x]$, because nothing can be a $T[x].UpperType[1]$ (..and a $T[x].UpperType[n]$) at the same time, which are the supertypes of $T[x]$. These supertypes are disjoint by definition as they don't share a common top supertype. Either you introduce a new top supertype, or you remove some of the supertypes of $T[x]$ }}}

3.2. *Pattern 2 (Exclusive Constraint between Types)*

In this pattern, subtypes of mutually exclusive supertypes (caused by an exclusive constraint) are detected. Figure 4 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.

Definition 2 For each exclusive constraint between a set of object-types $\{T_1, \ldots, T_n\}$, let $Subs(T_i)$ be the set of all possible subtypes of the object-type T_i, $1 \leq i \leq n$. For every i and j, let $X = Subs(T_i) \cap Subs(T_j)$, where $i \neq j$. If $X \neq \{\}$, then all objects-types in X cannot be satisfied.

Algorithm-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) {

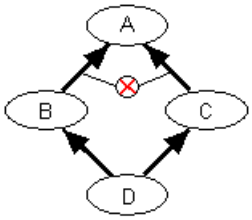

Fig. 4. Subtype with Exclusive Supertypes.

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].
X = X + (IntersectionOf(Exv[x].T[i].Subs, Exv[x].T[j].Subs))}}}
If (X is not empty) {
Satisfiability = false
Message=(Contradiction!! The concept(s) $X[1]$ (.., and $X[n]$) cannot be instantiated because of the exclusive constraint between (its/their) upper types. Either some subtype links should be dismissed, or the exclusive constraint should be remove.)}}

3.3. *Pattern 3 (Exclusive-Mandatory)*

In this pattern, contradictions between exclusion and mandatory constraints are detected. In Figure 5, we show three examples of unsatisfiable schemes. In the first case (a), the role r_3 will never be played. The mandatory and the exclusion constraints restrict that each instance of A must play r_1 and the instance that plays r_1 cannot play r_3. In the second case (b), both r_1 and r_3 will never be played. According to the two mandatory constraints, each instance of A must play both r_1 and r_3. At the same time, according to the exclusion constraints, an instance of A cannot play r_1 and r_3 together. Likewise, in the third case (c), r_3 and r_5 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 r_5, then this instance, which is also instance of A, cannot play r_1 or r_3. However, according to the mandatory constraint, each instance of A must play r_1 and, according to the exclusion constrain, it cannot play r_1, r_3 and r_5 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.

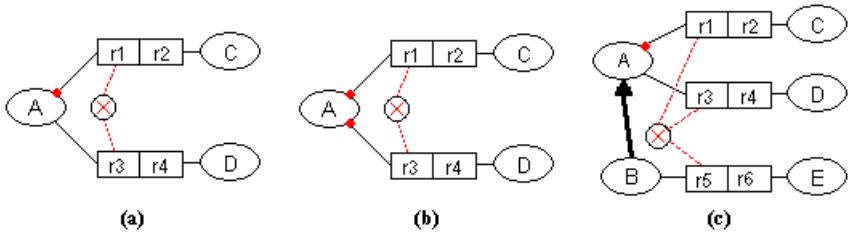

Fig. 5. Subtype with Exclusive Supertypes.

Definition 3 Given an exclusion constraint between a set of single roles $\{r_1, ..., r_n\}$. Let

T_i be the object-type connected to the role r_i (where $1 \leq i \leq n$); and, $Subs(T_i)$ be the set of all possible subtypes of T_i. For every (r_i, r_j) where $i \neq j$ and r_i is a *mandatory* role, if $(T_i = T_j)$ or $(T_j \in Subs(T_i))$ then r_j cannot be satisfied.

Algorithm-3: For each exclusion constraint Exs[x] between a set of single roles {
Let XRoles = the set of all roles participating in Exs[x].
Let ManRoles = the set of all mandatory roles in XRoles.
If (ManRoles is not empty)
For (i=1 to ManRoles.Size){
Let ManRoles[i].T = the object-type that plays the role ManRoles[i]
Let ManRoles[i].T.Subs = the set of all subtypes of ManRoles[i].T
For (j=1 to XRoles.Size) {
Let XRoles[j].T = the object-type that plays the role XRoles[j]
If ((ManRoles[i].T = XRoles[j].T) OR (In(XRoles[j].T, ManRoles[i].T.Subs)) AND (i ≠ j)
X = X + XRoles[j]} }
If (X is not empty){
Satisfiability = false
Order(X) //the super of the object types playing X first, then the mandatory roles, etc.
Message=(Contradiction!! between the exclusion and mandatory constraints. While the exclusion means that an instance of a $X[1].T$ that is $X[1]$ a $X[1].T2$ cannot be $X[2]$ a $X[2].T2$ (,.. and cannot be $X[n]$ a $X[n]$) at the same time, the Mandatory means that this instance must be $X[1]$ a $X[1].T2$ (,.. and must be $X[n]$ a $X[n].T2$). This implies that role(s) $X[1]$ (,.. and $X[n]$) will never be played. Either the exclusion or the mandatory constraints should be dismissed.)}}

3.4. *Pattern 4 (Frequency-Value)*

In this pattern, contradictions between value and frequency constraints are detected. In Figure 6, the role r_1 cannot be populated. If the frequency constraint FC$(3-5)$ on r_1 is satisfied, each instance of A must play r_1 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 $\{x_1, x_2\}$.

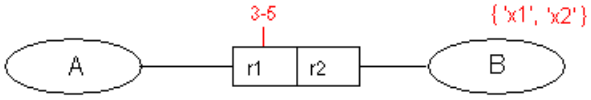

Fig. 6. Contradiction between value and frequency constraints.

Definition 4 For each fact-type (ArB), let c be the number of 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. If $c < n$, then r cannot be satisfied, as the value and the frequency constraints contradict each other.

Algorithm-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 T2.Values = the value constraint on the object-type related to T.
// if there is no value constraint on T, then T.Values = null
If (T2.Values is not null) and (T2.Values.size < F[x].min) {
Satisfiability = false.

Message=(Contradiction!! between the frequency and the value constraints. The value constraint implies that T should $T.r1$ at least $F[x].min$ different $T2$(s), but there are only $T2.Values.size$ possible values of $T2$. In other words, there is no enough $T2$(s) to fulfill the frequency constraint. Either the frequency or the value constraints should be changed or maybe dismissed.)}}

3.5. *Pattern 5 (Value-Exclusion-Frequency)*

In this pattern, contradictions between value, exclusion, and frequency constraints are detected. Figure 7 shows a particular contradiction between those three constraints. Due to the frequency constraint, there should be at least two different values to populate r_1. In order to populate r_3, we need, by the exclusion constraint, a value different from the two for role r_1. In total, we thus need three different values in order to be able to populate both r_1 and r_2, 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. A special case

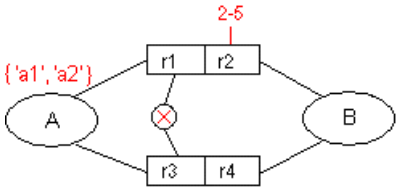

Fig. 7. Contradiction between value, exclusion, and frequency constraints.

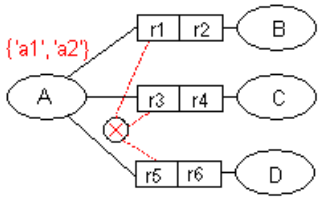

Fig. 8. Contradiction between value and exclusion constraints.

occurs in the absence of frequency constraints, e.g., Figure 8: according to the exclusion constraint, there should be at least 3 different values of A to play r_1, r_2 and r_3. However, according to the value constraint, there are only two possible values of A.

Remark: An exclusion constraint between n roles (where $n > 2$) can be split into $n(n-1)/2$ separate exclusion constraints between two roles (see [?]). For example, the exclusion constraint in Figure 8 is the *compact form* of three different exclusion constraints: between r_1 and r_3, r_3 and r_5, and r_1 and r_5. In this article, we assume that exclusion constraints are always in their most compact form.

Definition 5 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.

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.

Algorithm-5: For each exclusion constraint Exs[x] between a set of single roles {

Let XRoles = the set of roles participating in the exclusion Exs[x].
Let InvRoles = the set of inverse roles of XRoles.
For (1 to InvRoles.size) {
  If (InvRoles[i].frequency is not Null) F = F + InvRoles[i].frequency.min.
  else F = F+1; }
Let T = the object-type that plays all roles in XRoles.
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) {
  Satisfiability = false.
  If (T.Values is not null) Message =(Contradiction!! between the value, the exclusion, and the frequency constraints. In order to fulfill the frequency and the exclusion constraints, at least F different values of T are needed, However there are only $T.Values.size$ values of T are available according the value constraint. Either the value, the exclusion, and/or the frequency constraints should be changed.)
  else Message =(Contradiction!! between the value and the exclusion constraints. In order to fulfill the exclusion constraints, at least F different values of T are needed, However there are only $T.Values.size$ values of T are available, according to the value constraint. Either the value and/or the exclusion constraints should be changed.)}}

3.6. *Pattern 6 (Set-comparison constraints)*

In this pattern, contradictions between exclusion, subset, and equality constraints are detected. These three kinds of constraints are called set-comparison constraints. Figure 9 shows a contradiction between the exclusion and the subset constraints. This contradiction implies that both relations cannot be populated. The exclusion constraint between the two

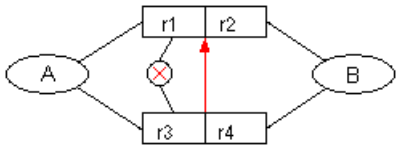

Fig. 9. Contradiction between subset and exclusion constraints.

roles r_1 and r_3 means that their populations should be distinct. However, in order to satisfy the subset constraint between the relations $(r1, r2)$ and $(r3, r4)$, the populations of r_1 and r_3 should not be distinct. In other words, the exclusion constraint between roles r_1 and r_3 implies an exclusion constraint between the relations $(r1, r2)$ and $(r3, r4)$ [?], which contradicts any subset or equality constraint between both predicates.

Figure 10 shows the implications for each set-comparison constraint that might be declared between roles or relations. These implications are taken into account when reasoning for contradictions between the three set-comparison constraints. In addition, an equality constraint is equivalent to two subset constraints opposing each other. Hence, we shall refer to a path of subset or an equality constraint as a *SubsetPath*. In the following we provide a formal definition of this pattern, which is divided into two definitions, one in case the exclusion constraint is declared between two roles, and the other between to relations.

Definition 6': For each exclusion constraint between two roles, there should not be any *Role-SubsetPath* between them, neither $(r_i \Rightarrow r_j)$ nor $(r_j \Rightarrow r_i)$. Otherwise, these roles are unsatisfiable. A *Role-SubsetPath* between two roles $(r_i \Rightarrow r_j)$ is a directed edge that is constructed as the following:

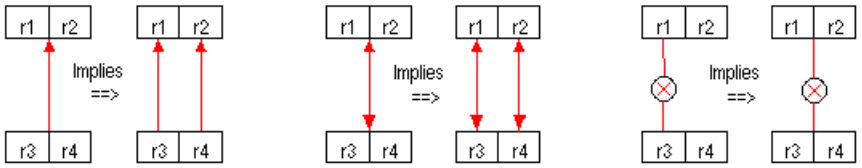

Fig. 10. Implications between the set-comparison constraints.

- If there is a subset constraint between them explicitly, such that $(r_i \rightarrow r_j)$; or implicitly through another role r_h, such that $(r_i \rightarrow r_h)$ and $(r_h \rightarrow r_j)$.
- If there is an explicit subset constraint between the relations including them, such that $(R_a(r_i, r_x) \rightarrow R_b(r_j, r_y))$; or implicitly through another relation $R_c(r_h, r_z)$, such that $(R_a(r_i, r_x) \rightarrow R_c(r_h, r_z))$ and $(R_a(r_h, r_z) \rightarrow R_b(r_j, r_y))$, where r_i, r_j and r_h have the same type.
- If there is an equality constraint between them explicitly, such that $(r_i \leftrightarrow r_j)$; or implicitly through another role r_h, such that $(r_i \leftrightarrow r_h)$ and $(r_h \leftrightarrow r_j)$.
- If there is an explicit equality constraint between the relations including them, such that $(R_a(r_i, r_x) \leftrightarrow R_b(r_j, r_y))$; or implicitly through another relation $R_c(r_h, r_z)$, such that $(R_a(r_i, r_x) \leftrightarrow R_c(r_h, r_z))$ and $(R_a(r_h, r_z) \leftrightarrow R_b(r_j, r_y))$, where r_i, r_j and r_h have the same type.

Definition 6": For each exclusion constraint between two relations, there should not be any *Relation-SubsetPath* between them, neither $(R_i \Rightarrow R_j)$ nor $(R_j \Rightarrow R_i)$. Otherwise, these relations (and thus their roles) are unsatisfiable. A *Relation-SubsetPath* between two relations $(R_i \Rightarrow R_j)$ is a directed edge that is constructed as the following:

- If there is a subset constraint between them explicitly, such that $(R_i \rightarrow R_j)$; or implicitly through another relation R_h, such that $(R_i \rightarrow R_h)$ and $(R_h \rightarrow R_j)$.
- If there is an equality constraint between them explicitly, such that $(R_i \leftrightarrow R_j)$; or implicitly through another relation R_h, such that $(R_i \leftrightarrow R_h)$ and $(R_h \leftrightarrow R_j)$.

Algorithm-6: For each exclusion constraint Exs[x] between roles or relations {
If (Exs[x] between relations) {
Let XRelations = the set of all relations participating in Exs[x].
//For each pair of relations participating in the exclusion
For (i = 1 to i = XRelations.size)
For (j = 1 to j = XRelations.size)
If (i $\neq$ j)
If (Relation-SubsetPath(XRelations[i], XRelations[j]) is not Null) {
Satisfiability = false.
Message=(Contradiction!! between the exclusion and the subset/equality constraints. In order to fulfill the exclusion constraint, the $XRelations[i].T$ that $XRelations[i]$ a $XRelations[i].T2$ must be different from the $XRelations[i].T$ that $XRelations[j]$ that $XRelations[j].T2$.However the subset/equality constraint implies that this $XRelations[i].T$ should be the same. Either the exclusion or the subset/equality constraints should be dismissed. }}
Else { // then the Exs[x] is between roles
Let XRoles = the set of all roles that are participating in Exs[x].
// For each pair of roles participating in the exclusion constraint
For (i = 1 to i = XRoles.size)

For (j = 1 to j = XRoles.size)
If (i $\neq$ j)
If (Role-SubsetPath(XRole[i], XRole[j]) is not Null) {
Satisfiability = false.
Message=(Contradiction!! between the exclusion and the subset/equality constraints. In order to fulfill the exclusion constraint, the $XRole[i].T$ that $XRole[i]$ a $XRole[i].T2$ must be different from the $XRole[i].T$ that $XRole[j]$ a $XRole[j].T2$. However the subset/equality constraint implies that this $XRole[i].T$ should be the same. Either the exclusion or the subset/equality constraints should be dismissed.}}}

3.7. *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 Figure 11 the uniqueness constraint indicates that the role r_1 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 r_1.

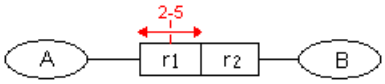

Fig. 11. Unsatisfiability of frequency and uniqueness constraint.

Definition 7 Given a role r with a frequency constraint FC$(n-m)$ and a uniqueness constraint on it at the same time, if $n > 1$ then r cannot be satisfied.

Algorithm-7: For each frequency constraint F[x] on a role{
Let F[x].min = the lower bound of the frequency constraint F[x].
Let r = the role on which the F[x] is placed.
If (r.uniqueness is not Null) and (F[x].min > 1) {
Satisfiability = false
Message= (Contradiction!! between the frequency and the uniqueness constraints. The frequency implies that $r.T$ should r at most $F[x].min$ and at least $F[x].max$ $r.T2$(s). However, the uniqueness implies that $r.T$ should r at most one $r.T2$. Either the frequency or the uniqueness constraints should be changed.)}

3.8. *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)$^{?,?}$. For example, Figure 12 shows an example of two ring constraints (symmetric and acyclic) placed on the *Manages* role. These constraints contradict each other because the symmetric implies that if person A manages person B, then person B also manages person A. However, the acyclic implies that a person cannot be directly (or indirectly through another person) a manager of himself.

The relationships between the six ring constraints are formalized by $^{?}$ using the Euler diagram as shown in Figure 13. 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.

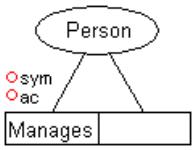

Fig. 12. the role $Manages$ is unsatisfiable

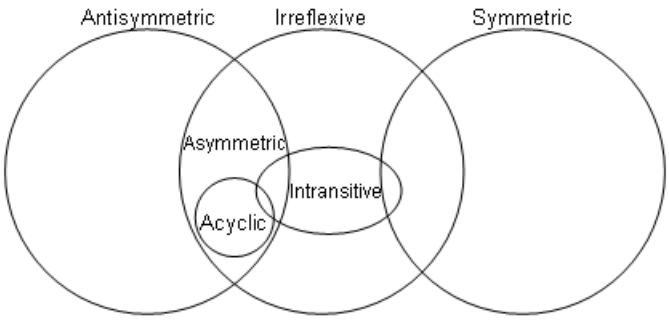

Fig. 13. Relationships between ring constraints.

In general, a role is unsatisfiable if it has two ring constraints, and these constraints are disjoint in the Euler diagram. Notice that there are only three combinations of ring constraint that are disjoint, and that lead to unsatisfiability, which are (symmetric, antisymmetric), (symmetric, asymmetric), and (symmetric, acyclic).

Definition 8. Given a role r with a set RC of ring constraints on it, if $(sym \in RC \wedge ans \in RC) \vee (sym \in RC \wedge as \in RC) \vee (sym \in RC \wedge as \in RC)$ then the role r cannot be satisfied.

Algorithm-8: For each role r with more than two ring constraint {
Let $r.RC$ = the set of ring constraints on r. Let $Temp$ be an empty set.
if ($sym \in r.RC$) {
if ($ans \in r.RC$) Temp.add("antisymmetric").
if ($as \in r.RC$) Temp.add("asymmetric").
if ($ac \in r.RC$) Temp.add("acyclic").
if ($Temp$ is not Null) {
Satisfiability = false.
Message= (Contradiction!! between the ring constraints. The symmetric implies that
if $r.T$ A r $r.T$ B, then $r.T$ B also r $r.T$ A. However, this symmetry contradicts the
$Temp[1]$ (.., $Temp[n]$) constraint(s). Either the symmetric or the other ring constraints
should be dismissed.) } } }

3.9. *Pattern 9 (Loops in Subtypes)*

As Subtypes in ORM are proper subtypes, the subtype relationship is acyclic. In other words, the population of a subtype is a strict subset (but not equal) of the population of its supertype[?], loops are illegal in ORM. Otherwise, one would have that a population is a strict subset of itself, which is not possible. In Figure 14, none of the object-types A, B, or

C can be satisfied since they form a loop.

Notice 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[h].

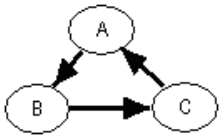

Fig. 14. Loop in subtypes.

Definition 9. Given a subtype T, let $Supers(T)$ be the set of all supertypes of T. If $T \in Supers(T)$, then the object-type T cannot be satisfied.

Algorithm-9: For each subtype T {
$T.Supers$ = the set of all super types of T.
If ($T \in T.Supers$) {
Satisfiability = false.
Message= (Contradiction!! There is a loop of subtypes, which implies, e.g. that T is a subtype of T. One of the subtype links in this loop should be removed.) }}}

4. Implementation: DogmaModeler and the CCFORM case study

This section illustrates the DogmaModeler's implementation of all patterns in interactive modeling. As an empirical usability experiment, we report our experience in applying the patterns during the development of the customer complaint ontology, and the lessons learned from the lawyers who built this ontology. At the end of this section, we compare this pattern-based reasoning approach with the description logic-based reasoning that we also support in DogmaModeler, and we argue that *the two approaches complement each other from a methodological viewpoint*.

DogmaModeler is a software tool for modeling and engineering ontologies. The philosophy of DogmaModeler is to enable non-IT experts to model ontologies with little or no involvement of an ontology engineer. This challenge is tackled in DogmaModeler through well-defined methodological principles: the double-articulation[i] and the modularization[j]

[h] Furthermore, unlike ORM, subtypes in most of the specification languages (e.g. OWL) are not proper subtypes, thus loops imply equality of the involved concepts.

[i] The *ontology double-articulation principle* suggests that an application axiomatization should be build in terms of (i.e. commits to) a domain axiomatization. While a domain axiomatization focuses on the characterization of the intended meaning (i.e. intended models) of a vocabulary at the domain level, application axiomatizations mainly focus on the usability of this vocabulary according to certain application/usability perspectives. An application axiomatization is intended to specify the legal models (a subset of the intended models) of the application(s) interest. For simplicity, one can imagine WordNet as a domain axiomatization, and ORM schema as an application axiomatization, where all terms/object-types in the schema are linked with terms/synsets in WordNet. The idea here is to enable: reusability of domain knowledge and usability of application knowledge, interoperability of applications, etc. See [?,?] for more details.

[j] The *ontology modularization principle* suggests that application axiomatizations be built in a modular manner. Application axiomatizations (e.g. ORM schemes) should be developed as a set of small modules and later composed to form, and be used as, one modular axiomatization. DogmaModeler implements a well-defined composition operator for automatic composition of modules. It combines all axioms introduced in the composed modules. See [?,?] for more details.

principles. Furthermore, DogmaModeler supports the use of ORM as a graphical notation for ontology modeling; the verbalization of ORM diagrams into pseudo-natural language (supporting flexible verbalization templates for 11 human languages, including English, Dutch, German, French, Spanish, Arabic, Russian, etc.) that allows non-experts to check, validate, or build ontologies; the automatic composition of ontology modules, through a well-defined composition operator; the incorporation of linguistic resources in ontology engineering; the automatic mapping of ORM diagrams into the DIG description logic interface and reasoning using Racer[k]; and many other functionalities.

Ontology validation in DogmaModeler is made simple and user friendly. Although users can use Racer for complete formal reasoning on ORM diagrams, also non-IT/logic experts are supported with an easy to understand reasoning approach (using the patterns), which explain them constraint contradictions, the cause of these contradictions, and suggestions of how to resolve these contradictions. DogmaModeler implements all patterns

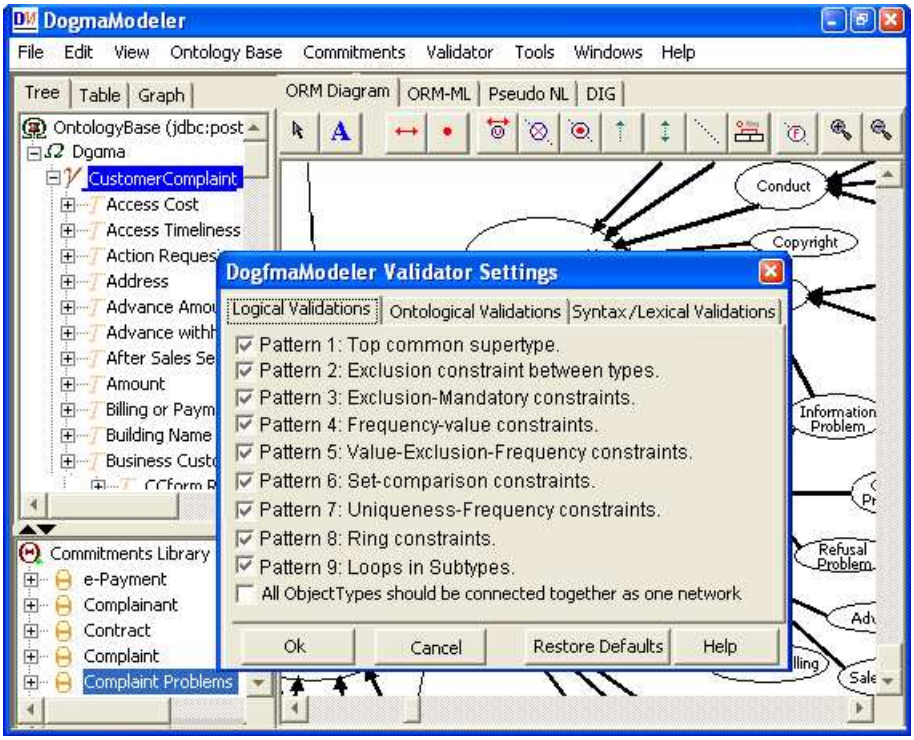

Fig. 15. DogmaModeler's support of patten-based reasoning.

described in this paper. Figure 15 displays these patterns as a menu in the DogmaModeler Validator Settings window. Users can choose to enable or disable the enforcement of these validation patterns when reasoning about the satisfiability of an ORM schema. The DogmaModeler typically implements the satisfiability algorithms that we have presented in section 3.One can see, from these algorithms, that not only unsatisfiability is detected, but also, some details about the detected problems (through the generated message), including which constraints cause the unsatisfiability, the problems with the other constraints, and

[k]Racer is a description logic based reasoning engine.

some suggestions to resolve the problem. Figure 16 and 17 illustrate unsatisfiable ORM diagrams and the result of the reasoning. When a user reason about an ORM schema, DogmaModeler first runs the pattern-based approach, if the schema passes all patterns and no problems are catched, then it runs the DL-based approach (using Racer) for completeness. DogmaModeler calls these patterns not only while a modeling mistake is made by a on-

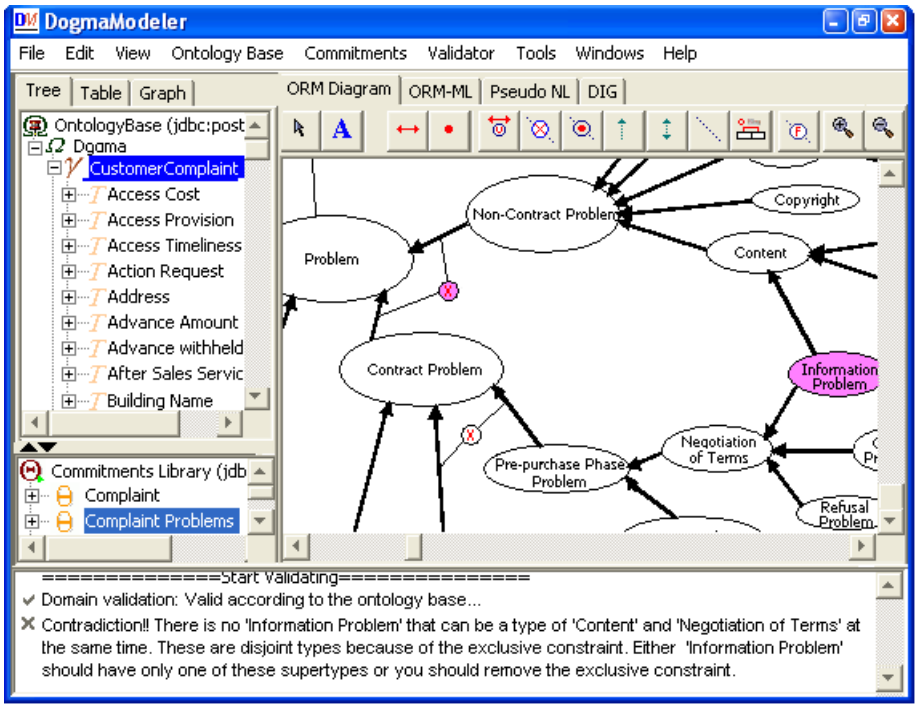

Fig. 16. An example of a satisfiable schema from the CCFORM project.

tology modeler, but also when an ORM diagram is composed with another. As we shall report shortly, the main source of unsatisfiability problems is when two incompatible ORM diagrams are composed with each other.

Although it is developed as a research prototype[?] DogmaModeler has been used in many projects for ontology and business rule modeling, including CCFORM, FFPOIROT, InnovaNet, and SCOPE. In the following we present our experience in using DogmaModeler (focusing on the application of the pattern-based reasoning) in the CCFORM project.

4.1. *The CCFORM case study*

CCFORM is an EU funded project (IST-2001-38248) with the aim of studying the foundation of a central European customer complaint portal. The idea is that any consumer can register a complaint against any party about any problem in one portal. This portal should: support 11 languages, be sensitive to cross-border business regulations, dynamic, and can be extended by companies. To manage this dynamicity and to control companies' extensions, a customer complaint ontology (CContology) has to be built as the basis of the CC portal. In other words, the complaint forms are generated based on the ontology[1].

[1]See [?] for DogmaModeler's support of generating web forms automatically out of a given ORM schema.

The CContology[?] comprises classifications of complaint problems, complaint resolutions, complainant, complaint-recipient, "best-practices", rules of complaint, etc. The main uses of this ontology are 1) to enable consistent implementation (and interoperation) of all software complaint management mechanisms based on a shared background vocabulary, which can be used by many stakeholders. 2) to play the role of a domain ontology that encompasses the core complaining elements and that can be extended by either individual or groups of firms; and 3) to generate CC-forms based on its ontological commitments and to enforce the validity (and/or integrity) of their population. More information about the CContology, including the ontology content itself, can be found in [?,?,?].

The CContology is developed in 11 human languages, and it is a result of six groups (about 55 experts), including lawyers, consumer-affairs and e-business experts. None of these experts was really aware of what an ontology is or how it can be used. Our role in the project was to lead the ontology (and the multilingual) engineering tasks. We trained about 10 of those experts on how to use DogmaModeler and build ontologies. The training was done in two sessions, each of 3 hours. We basically focused on explaining the ORM notation that was easily understood. We have found that the verbalization of the ORM rules was a great mechanism to communicate with these experts. To enable collaborative development (and also improve the reusability), the CContology was developed as a set of 7 modules. Each module focuses on a certain subject matter, such as classifications of complaint problems, classifications of complaint resolutions, etc.

Although the CContology is a medium size ontology[m], it illustrates the value of the pattern-based reasoning in interactive modeling. We have found that the experts did many modeling mistakes (i.e. constraint contradictions) at the beginning, but the final version of ontology did not contain any contradiction indeed. As constraint contradictions are detected in DogmaModeler during the modeling process and in an interactive manner, we found that this interaction is a self-learning mechanism. In other words, 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.

The source of these mistakes was mostly due to the lack of ORM understanding, which is natural in our opinion. Figure 17 is an example of such a mistake. In this diagram, the lawyers intended to model that a complainant cannot register a complaint against himself, so instead of placing an exclusion constraint between the two relationships registers/made_by and receives/made_against, they place it between the two *roles* made_by and made_against. This mistake was discovered by DogmaModeler as it contradicts the mandatory constraints (see pattern 3). The second source of contradictions is the autonomous development of modules. When composing modules, the resulted composition might be unsatisfiable due to contradicting constraints among these modules. This is true although each module individually is satisfiable. Figure 16 was an example of this mistake. In one module the 'Information Problem' was a subtype of 'content' and in the second module it was also a subtype of 'negotiation of terms'. The resulted composition of these two modules, where 'Information Problem' became subtype of two disjoint concepts, which generates unsatisfiability.

The coverage of the patterns for the CContology was sufficient. Although the lawyers did many modeling mistakes but all of these mistakes were discovered by DogmaModeler. In fact, we found only 4 patterns were violated, which are: pattern 1, pattern 2, pattern 3, and pattern 4. Pattern 2 was the most violated pattern in general. However, we also found

[m] about 220 concepts and 300 relations

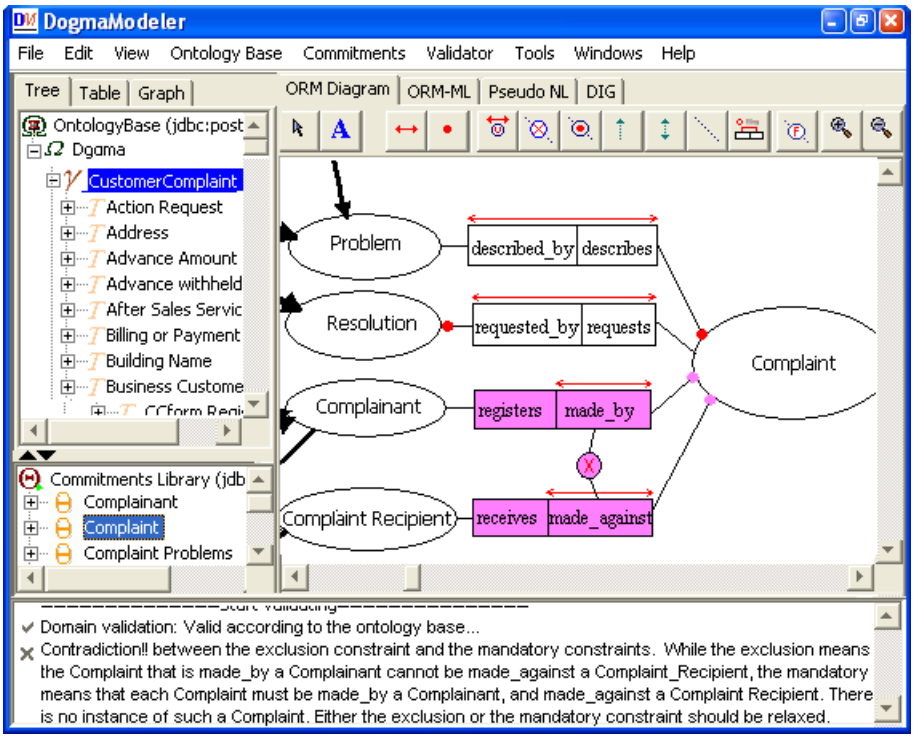

Fig. 17. Another example of a satisfiable schema from the CCFORM project.

that Pattern 3 was the most violated pattern during the modeling processes, and pattern 2 was still the most violated during the composition. Pattern 4 was violated only two times, which is due to limited use of the value constraint in the CContology.

The messages that DogmaModeler generates when a problem is discovered was understandable. Although such messages cannot always be grammatically correct, as they depend mostly on the linguistic labels used the schema, but we did not face any unclear message. The suggestions provided in each message -on how to resolve the contradiction- were always correct, from a logic viewpoint. However, from a methodological viewpoint, one cannot really know what is the best solution to resolve the problem. For example, although the message in figure 17 says "Either the exclusion or the mandatory constraint should be relaxed", which is true from a formal viewpoint, but the real reason of this contradiction was the incorrect placing of the constraint. For such reasons, we try to make the message as clear and expressive as possible so that users can spot (i.e. between the lines) their real mistakes.

4.2. *Pattern-based verses DL-based reasoning in DogmaModeler*

As we have mentioned earlier, we have also tackled the ORM satisfiability problem in another way. We have mapped ORM into both the $\mathcal{DLR}^?$ and the $\mathcal{SHOIN}$/OWL$^?$ description logics, which are powerful and decidable fragments of first order logic. Based on these mapping, DogmaModeler maps$^?$ ORM diagrams automatically into DIG, which is a description logic interface (XML-based language) that most reasoners (such as Racer, FaCT++, etc) support. DogmaModeler is integrated with the Racer description logic rea-

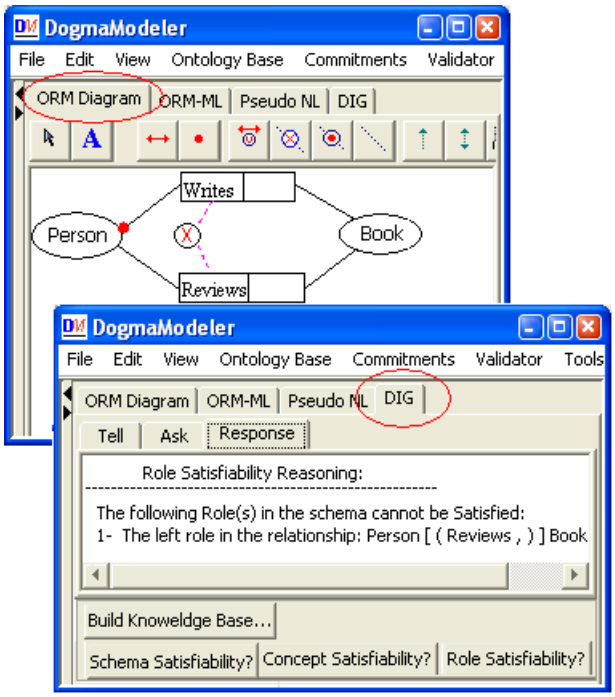

Fig. 18. DogmaModeler's support of description logic-based reasoning.

soning server, which acts as a background reasoning engine. Figure 16 shows a screen shot of this implementation. The first window shows an ORM diagram, while the second window shows the reasoning results on this diagram. The results indicate that the role Reviews cannot be satisfied.

This mapping[n] into description logics provides 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 DL knowledge base.

When a user reason about an ORM schema, DogmaModeler first calls the pattern-based approach, if all patterns passed and no problems are catched, then DogmaModeler calls Racer (the DL-based approach) for complete reasoning.

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 and cheap to implement, which allows speedy reasoning in interactive modeling tools. More importantly is that unsatisfiability/contradiction messages can be customized for the ease of non-IT domain experts, explaining the sources of the unsatisfiability, and offering solutions to these contradictions. On the other side, description logic based reasoning offer complete solutions, so that it is guaranteed at least theoretically that there are no undiscovered contradictions. However, these complete solutions cannot offer more than a list of the unsatisfiable concepts. In DogmaModeler, even in the presence

[n] Some ORM constrains cannot be mapped into description logic or they are not yet supported by any DL reasoner, such as external uniqueness, ring, multiple-role frequency, etc. Please refer to [?,?,?] for the details.

of a complete reasoner, the patterns are used to quickly detect any "trivial" inconsistencies, before calling the more expensive (but complete) procedure.

In the next section we discuss some efforts by other researchers who tried to modify the internals of DL reasoners for generating unsatisfiability messages in an easy to understand way and to propose solutions.

5. Related Work

Related Work in description logics

As we have mentioned before, the need to trace the causes of unsatisfiability and conflicts has recently been realized by several research communities. In the context of debugging OWL ontologies, a group from the university of Manchester has developed a software[?] based on heuristics for identifying the sources of unsatisfiability for OWL ontologies. This software is based on a list of common errors and mistakes that have been gathered[?] by a group of researchers during tutorials and courses. The authors claimed that this system is independent of any particular reasoner, they call it "Black Box".

A "Glass Box" approach to debug OWL ontologies and diagnose inconsistencies has been proposed[?] by a group from the university of Maryland. This approach suggests to modify the internals of the Pellet reasoner, which is developed by the same group. This approach offers browse-able messages to help users identify the sources of the inconsistencies themselves. The authors of this approach explained the fundamental challenge of computing the sources of unsatisfiability for an unsatisfiable ontology, and concluded that only some situations can be improved, which are: "inconsistency of assertions about individuals", "individuals related to unsatisfiable concepts", "defects in class axioms involving nominals". Related to this approach, another methods[?,?] have been proposed to explain some reasoning on ontologies, but these approach are mainly concerned with explaining subsumptions. It also worth to note that identifying the causes of conflicted is becoming an important goal in evolution management[?,?]. However these approaches are not in its infancy and are not concerned with explaining messages to users.

Compared with our approach, we have found that none of the above mentioned is concerned with providing suggestions to resolve unsatisfiability. Furthermore, as the screenshots, from the above cited articles show, the results are not convenient for a non-IT domain experts. In terms of completeness, neither our approach, nor the above approaches can be complete, which is due the nature of the problem. In addition, one cannot also compare the comprehension of the approaches to each other, because of the disparities between the OWL and the ORM constructs. For example, many of the common errors in OWL[?] are considered syntax violations and thus detected by the ORM parser before any reasoning.

Related Work in ORM

In [?], 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 section 3.

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[o] 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

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

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 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 (see Figure 19). For example, populate r_5 with some 'a', then 'a', by the sub-

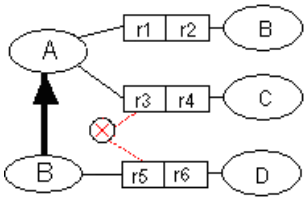

Fig. 19. ORM schema violating formation rule 6 but with satisfiable roles.

typing, must play one of the roles r_1 or r_3. It cannot play r_3 due to the exclusion constraint but nothing keeps it from playing r_1.

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[p]. 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 Section 3, focus on the semantical aspect of unsatisfiability only.

In [?], the RIDL-A module of the RIDL* workbench, a database engineering tool based on the NIAM methodology [?], 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.

[p]Maximal cardinalities in [?] correspond to value constraints.

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 [q]. 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 [?], i.e., a role r_1 is a subset of r_2 if every element playing role r_1 also plays r_2. In particular, r_1 does not need to be a strict subset of r_2: 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[r] 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.

6. Conclusions and Further Research

We presented a pattern-based reasoning approach that offers a user-friendly reasoning mechanism. 9 patterns of constraint contradictions that lead to unsatisfiability in ORM models are identified. Not only detecting constraint contradictions, but also focused on providing a clear explanation about the detected contradictions, the causes, and suggestions on how to resolve these contradictions.

We illustrated the DogmaModeler's implementation of all patterns in interactive modeling. As an empirical usability experiment, we reported our experience in applying the patterns during the development of a customer complaint ontology. We compared this pattern-based reasoning approach with the description logic-based reasoning that we also support in DogmaModeler, and we argued that the two approaches complement each other from a methodological viewpoint.

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.

One may notice that our patterns can be easily translated to other knowledge representation languages, especially for ontology and business-rules modeling tools. We plan to apply these patterns for reasoning on OWL ontologies[s].

Acknowledgment: We are indebted to Pieter De Leenheer, Robert Meersman and Olga De Troyer for their suggestions during this research. We wish to thank specially the anonyms reviewers of this article who give us many ideas and very helpful suggestions that turn the paper to be more precise and mature. This research is partially supported by the SEARCHiN (MTKD-CD-2006-042467) and the Knowledge Web (IST-2004-507482) projects.

[q] A constraint is superfluous -or implied- if it can be derived from other constraints.

[r] The OTSET of a role corresponds, roughly, to the population of a role.

[s] One of our master students aims to implement these patterns in Protg (an ontology modeling tool), as part of his thesis on interactive ontology modeling.