[J05a] Jarrar, M.: Modularization and automatic composition of Object-Role Modeling (ORM) schemes. In Halpin, T., Meersman, R.: Proceeding of the International Workshop on Object-Role Modeling (ORM'05), OTM 2005 Workshops. LNCS, Vol., Springer. 2005.

Modularization and automatic composition of Object-Role Modeling (ORM) schemes

Mustafa Jarrar

Vrije Universiteit Brussel, Brussels, Belgium
mjarrar@vub.ac.be

Abstract. In this paper we present a framework and algorithm for modularization and composition of ORM schemes. The main goals of modularity are to enable and increase reusability, maintainability, distributed development of ORM schemes. Further, we enable effective browsing and management of such schemes through libraries of ORM schema modules. For automatic composition of modules, we present and implement a composition operator: all atomic concepts and their relationships (i.e. fact-types) and all constraints, across the composed modules, are combined together to form one schema (called modular schema).

Keywords: Object Role Modeling, ORM, NIAM, Conceptual Modeling, Ontology, Formal ontology engineering, DOGMA, DogmaModeler, Modularization, Composition, Reusability, Distributed Development, Maintainability.

1 Introduction and motivation

ORM (Object-Role Modeling) [H01] is a conceptual modeling approach that was developed in the early 70's. It is a successor of the NIAM (Natural-language Information Analysis Method) [VB82]. The ORM conceptual schema methodology is fairly comprehensive in its treatment of many "practical" or "standard" business rules and constraint types (e.g. identity, mandatory, uniqueness, subsumption, subset, equality, exclusion, value, frequency, symmetric, intransitive, acyclic, etc.). Furthermore, ORM has an expressive and stable graphical notation since it captures many rules graphically and it minimizes the impact of change on the models.

Although ORM was originally developed as a database modeling approach, it has been also successfully reused in other conceptual modeling scenarios, such as XML-Schema conceptual design [BGH99], business rule modeling language [H04][N99][DJM02a], ontology modeling [JDM03][J05], etc. Hence, we shall regard an ORM schema, in this paper, as a general conceptual model independently of a certain application or modeling scenario; and we sometimes interchange the term “ORM schema” with the term “axiomatization” to refer to the same thing.

The main idea of ORM modularization in this paper is to decompose an ORM schema into a set of smaller related modules, which: 1) are easier to reuse in other kinds of applications; 2) are easier to build, maintain, and replace; 3) enable distributed development of modules over different locations and expertise; 4) enable the effective management and browsing of modules, e.g. enabling the construction of libraries of ORM modules [JM02b][1].

To compose modules automatically, we propose a composition operator: all atomic concepts and their relationships (i.e. fact-types) and all constraints, across the composed modules, are combined together to form one schema (called *modular schema*).

1.1 A simple example

In what follows, we give an example to illustrate the (de)composition of ORM schemes. Fig. 1 shows two ORM schemas of Book-Shopping and Car-Rental applications. Notice that both schemes share the same axioms about payment.

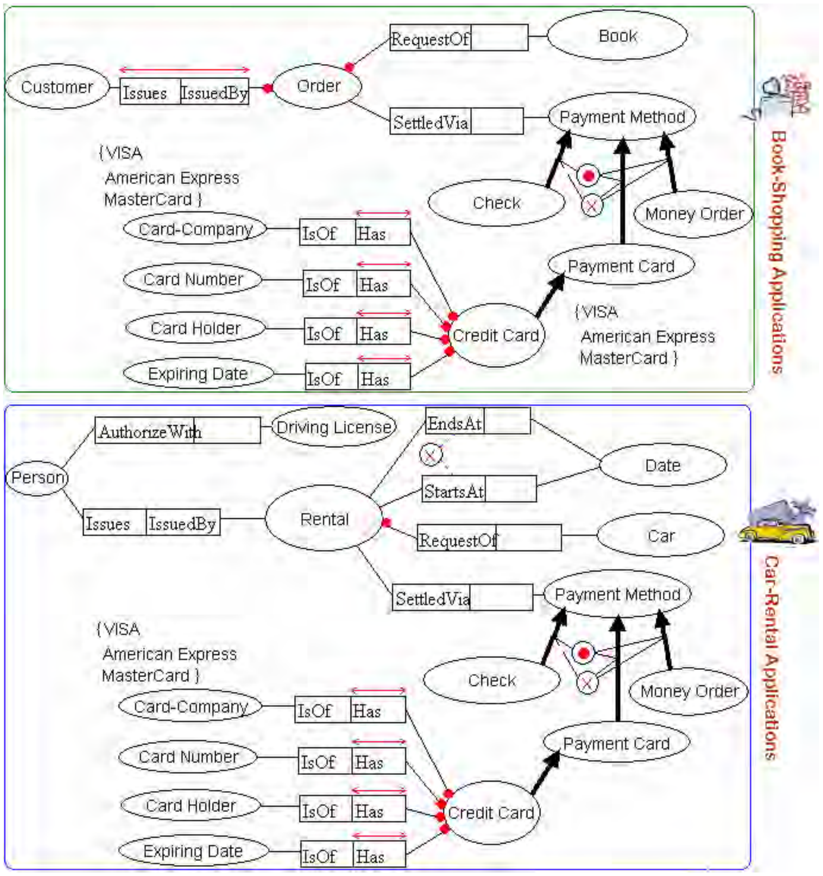

Fig. 1. Book-shopping and Car-Rental schemes.

[1] Notice that reusability, maintainability, and distributed development of ORM schemes might not be challenges in database modeling (the original usage of ORM), but they are urgent demands when using ORM e.g. in ontology Engineering [J05][SMS+05].

Instead of repeating the same effort to construct the axioms of the “payment” part, we suggest decomposing these schemes into three modules, which can be shared and reused among other applications (see fig. 2). Each application-type (*viz.* Book-Shopping and Car-Rental) selects appropriate modules (from a library) and composes them through a composition operator. The result of the composition is seen as one schema[2].

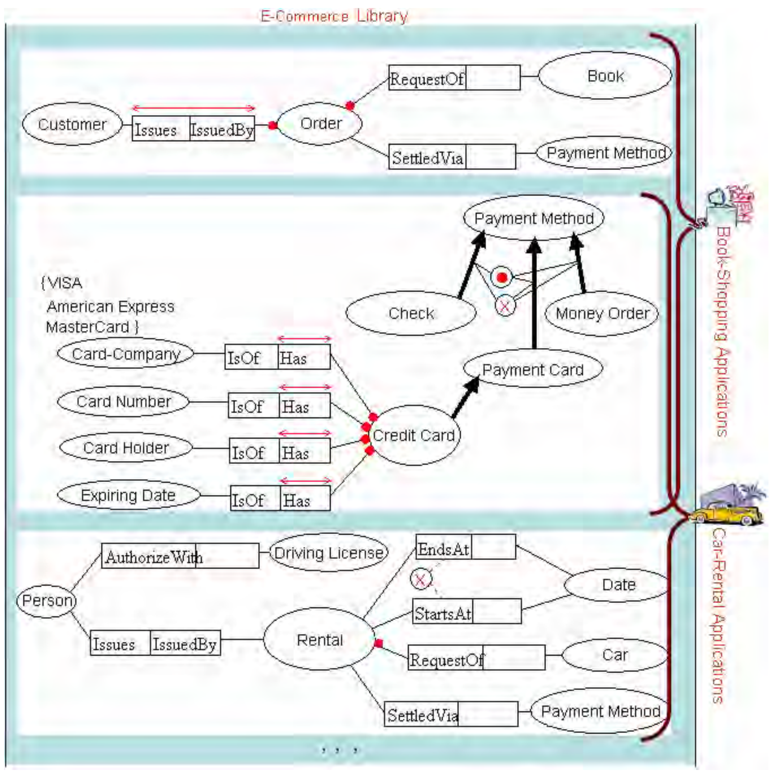

Fig. 2. Modularized schemes.

Engineering schemes in this way will not only increase their reusability, but also the maintainability and management of these axiomatizations[3]. As the software engineering literature teaches us, small modules are easier to understand, change, and replace [P72] [SWCH01]. An experiment by [BBDD97] proves that the modularity of object-oriented design indeed enables better maintainability and extensibility than structured design. Decomposing schemes into modules also enables the distributed development of these modules over different location, expertise, and/or stakeholders.

[2] The illustrated composition in this example is very simplistic, as each pair of modules overlap only in one object-type, i.e. the “Payment Method”. In farther sections, we discuss more complicated compositions, in which rules in different modules may contradict or imply each other.

[3] In this way, one can imagine axiomatizations (/schemes) as large sets of business rules modularized and organized as sets of compose-able modules.

As an analogy, compare the capability of distributing the development of a program built in Pascal with a program built in JAVA, i.e. structured verses modular distributed software development.

The modularity criteria could typically be subject-oriented and/or purpose/task-oriented. Subject-oriented parts should be released into separate modules, e.g. separate between the financial axioms (e.g. salary, contract, etc.) and the academic axioms (e.g. course, exams, etc.). The general purpose/task-oriented parts of an axiomatization could be released into separate modules, e.g. the axiomatization of “payment”, “shipping”, “invoicing”, which are often repeated in many e-commerce applications.

2 Composition Framework

To compose modules we define a composition operator. All concepts and their relationships (i.e. fact-types) and all constraints, across the composed modules, are combined together to form one axiomatization. In other words, the resultant composition is the union of all axioms in the composed modules. As shall be discussed later, a resultant composition might be *incompatible* in case this composition is not satisfiable, e.g. some of the composed constraints might contradict each other.

Our approach to composition is constrained by the following argument. A developer, when including a module into another, must expect that all rules in the included module are inherited by the including module, i.e. *all axioms in the composed modules must be implied in the resultant composition*. Formally speaking, the set of possible models for a composition is the intersection of all sets of possible models for all composed modules. In other words, we shall be interested in the set of models that satisfy all of the composed modules.

In fig. 3, we illustrate the set of possible instances (i.e. possible models) for a concept constrained differently in two modules composed together. Fig. 3(a) shows a compatible composition where the set of possible instances for M.c is the intersection of the possible instances of M_1.c and M_2.c. Fig. 3(b) shows a case of incompatible composition where the intersection is empty.

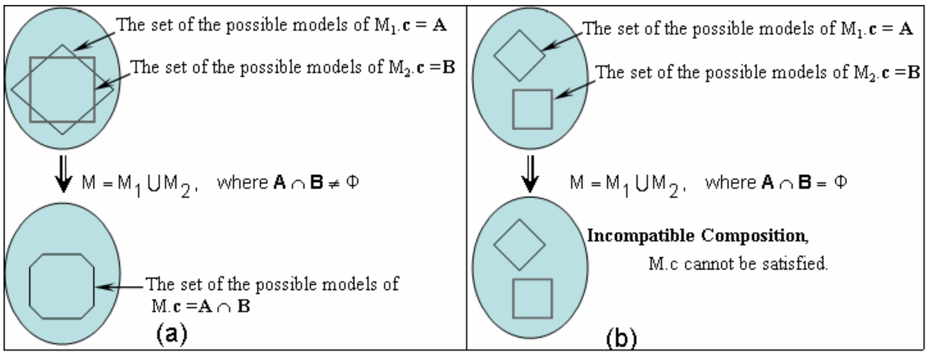
The set of the possible models of M_1.**c** = **A**
The set of the possible models of M_2.**c** =**B**
$M = M_1 \cup M_2$, where $\mathbf{A} \cap \mathbf{B} \neq \Phi$
The set of the possible models of M.**c** =**A** ∩ **B**
(a)
The set of the possible models of M_1.**c** = **A**
The set of the possible models of M_2.**c** =**B**
$M = M_1 \cup M_2$, where $\mathbf{A} \cap \mathbf{B} = \Phi$
Incompatible Composition,
M.c cannot be satisfied.
(b)

Fig. 3. (a) Compatible composition, (b) Incompatible composition.

Notice that our approach to module composition is not intended to integrate or unite the extensions (i.e. ABoxes) of a given set of modules, as several approaches to schema integration aim to do [SP94] [SK03][BS03]. *Our concern is to facilitate developers (at the development phases) with a tool to inherit (or reuse) axiomatizations without "weakening" them*. In other words, when including a module into another module (using our composition operator, which we shall formalize in the next sections) all axioms defined in the included module should be inherited by (or applied in) the including module.

It is also worth to mention that Vermeir [V83] has proposed an approach for modularizing large ORM diagrams based on heuristic procedures. However, this approach is not related to ours, as it is only concerned with how to "view" a one large ORM diagram in different degrees of abstraction or viewpoints. Another similar approach is proposed by Shoval [S85].

2.1 Definition (Module) A module is an axiomatization (i.e. a typical ORM Schema) of the form M = <**P**, **Ω**>, where **P** is a non empty set of fact-types, i.e. the set of object-types and their relationships; **Ω** is a set of constraints which declares what should necessarily hold in any possible world of M. In other words **Ω** specifies the legal models of M.

2.2 Definition (Model, Module satisfiability) Using the standard notion of an interpretation of a first order theory, an interpretation I of a module M, is a *model* (also called "legal model") of M iff each sentence of M (i.e. each $\rho \in$ P and each $\omega \in \mathbf{\Omega}$) is true for I.

Each module is assumed to be self-consistent, i.e. satisfiable. Module satisfiability demands that each role in the module can be satisfied [BHW91]. For each ρ in a given module M, ρ is satisfiable w.r.t. to M if there exists a model I of M such that $\rho^I \neq \emptyset$.

Notice that we adopt a strong requirement for satisfiability, as we require each role in the schema to be satisfiable. A weak satisfiability requires only the module itself (as a whole) to be satisfiable [H89][BHW91].

2.3 Definition (Composition operator) Modules are composed by a composition operator, denoted by the symbol '$\oplus$'. Let M = $M_1 \oplus M_2$, we say that M is the composition of M_1 and M_2. M typically is the union of all fact-types and constraints in both modules. Let $M_1 = <\mathbf{P}_1, \mathbf{\Omega}_1>$ and $M_2 = <\mathbf{P}_2, \mathbf{\Omega}_2>$, the composition of $(M_1 \oplus M_2)$ is formalized as $M = < \mathbf{P}_1 \oplus \mathbf{P}_2, \mathbf{\Omega}_1 \oplus \mathbf{\Omega}_2>$. A composition $(M_1 \oplus M_2)$ should *imply* both M_1 and M_2. In other words, for each model that satisfies $(M_1 \oplus M_2)$, it should also satisfy each of M_1 and M_2. Let $(M_1)^I$ and $(M_2)^I$ be the set of all possible models of M_1 and M_2 respectively. The set of possible models of $(M_1 \oplus M_2)^I = (M_1)^I \cap (M_2)^I$. A composition is called *incompatible* iff this composition cannot be satisfied, i.e. there is no model that can satisfy the composition, or each of the composed modules.

2.4 Definition (Modular schema) A modular schema $M = \{M_1 \ldots M_n, \oplus\}$ is a set of modules with a composition operator between them, such that $P = (P_2 \oplus \ldots \oplus P_n)$ and $\mathbf{\Omega} = (\mathbf{\Omega}_1 \oplus \ldots \oplus \mathbf{\Omega}_n)$.

3 Composition of ORM conceptual schemes

In this section we present an algorithm for automatic composition of modules specified in ORM. We adopt the ORM formalization and syntax as found in [H89][H01], excluding three things. First, although ORM supports n-ary predicates, only binary predicates are considered in our approach. 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.

A composition of two modules ($M = M_1 \oplus M_2$) is performed in the following steps: 1) Combine the two sets of fact types ($\mathbf{P} = \mathbf{P}_1 \oplus \mathbf{P}_2$). 2) Combine the two sets of constraints, $\mathbf{\Omega} = \mathbf{\Omega}_1 \oplus \mathbf{\Omega}_2$. 3) Reason to find out whether the composition is satisfiable. Optionally, 4) reason to eliminate all implied constraints from the composition. The last two steps are not presented in this paper because of the limited space. See our approach in [JH05] for reasoning about the satisfiability of ORM Schemes. For step 4 we refer to [H89] for a comprehensive specification of constraint implication in ORM.

The composition is considered an incompatible operation (and thus terminated) iff the result cannot be satisfied.

Step 1: combining fact types

When composing two sets of fact-types ($\mathbf{P} = \mathbf{P}_1 \oplus \mathbf{P}_2$), an object-type $M_1(T)$ in module M_1 and a object-type $M_2(T)$ in module M_2 are considered exactly the same concept iff they are referred to by the same term T, and/or URI. Formally, ($M_1(T) = M_2.(T)$). Likewise, two fact-types are considered exactly the same ($M_1.\langle T_1, r, r', T_2\rangle = M_2.\langle T_1, r, r', T_2\rangle$) iff $M_1(T_1) = M_2(T_1)$, $M_1.(r) = M_2.(r)$, $M_1.(r') = M_2.(r')$, and $M_1.(T_2) = M_2.(T_2)$[4]. See fig. 4.

In case that M_1 and M_2 do not share any object-type between them (i.e. two disjoint sets of fact-types), the composition ($M_1 \oplus M_2$) is considered an incompatible operation[5], as there is no model that can satisfy both M_1 and M_2. Notice that in case an object-type is specified as “lexical” in one module and as “non-lexical” in another (e.g. ‘Account’), then in the composition, this object-type is considered “non-lexical”.

[4] T refers to a Term (concept label), r refers to a role, r’ refers to an inverse role.

[5] In practice, we weaken this requirement to allow the composition of disjoint modules. For example, in case one wishes to compose two disjoint modules and later compose them within a third module that results in a joint composition.

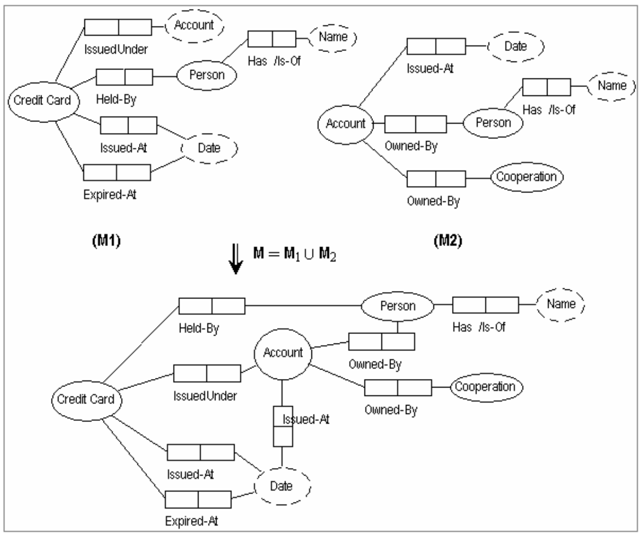

Fig. 4. Combining ORM fact types.

Step 2: combining constraints

When composing two modules, the *combination* of all constraints ($\Omega_1 \oplus \Omega_2$) should be syntactically valid according to the ORM syntax. For example, some constraints need to be syntactically combined into one constraint. *The combination of a set of constraints should imply all of them*. Furthermore, some logical (i.e. satisfiability and implication) validations are also performed in this step, e.g. in case of combining two constraints that contradict or imply each other. In the following, we show how all ORM constraints can be combined.

Step 2.1: Combining value constraints

Given two value constraints $T.v_1$ and $T.v_2$ on the same object-type T, (notice that v_1 and v_2 are two sets of values), their combination is the intersection $T.v = T.v_1 \cap T.v_2$, see fig. 5(a). If $T.v_1 \cap T.v_2$ is empty, then the composition ($M_1 \oplus M_2$) is considered as incompatible operation, because the value constraints contradict each other and thus the object type cannot be satisfied, see fig. 5(b).

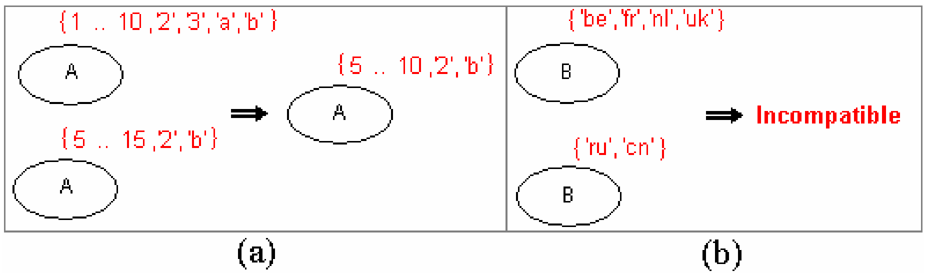

Fig. 5. Combining value constraints.

Step 2.2: Combining mandatory constraints

When composing two modules, all mandatory constraints are included in the composition without any specific combining operation.

Step 2.3: Combining disjunctive mandatory

When composing two modules, all disjunctive mandatory constraints are included in the composition without any specific combining operation.

Step 2.4: Combining uniqueness and frequency constraints

When composing modules, uniqueness and frequency constraints are combined as follows:

- As internal uniqueness implies predicate uniqueness [H89], the combination of these two constraints is internal uniqueness (see fig. 6. (a) and (b)).
- In case of internal uniqueness and frequency constraints on the same role (see fig. 6(c)), the composition of ($M_1 \oplus M_2$) is considered an **incompatible operation**, because the two constraints contradict each other [H89], and thus the role cannot be satisfied. Recall that a frequency of maximum 1 is considered internally uniqueness (see fig. 6(d)).
- In case of two frequency constraints on the same role, FC_1(min-max) and FC_2(min-max), the combination FC(min-max) is calculated as FC.min = MaxOf(FC_1.min, FC_2.min) and FC.max = MinOf(FC_1.max, FC_2.max), see fig. 6(e). In case the FC.min > FC.max, see fig. 6(f), then the composition of ($M_1 \oplus M_2$) is considered an **incompatible operation**, because the two constraints are in conflict each other, and the role cannot be satisfied.
- In other cases, all constraints are included in the composition without any specific combining operation.

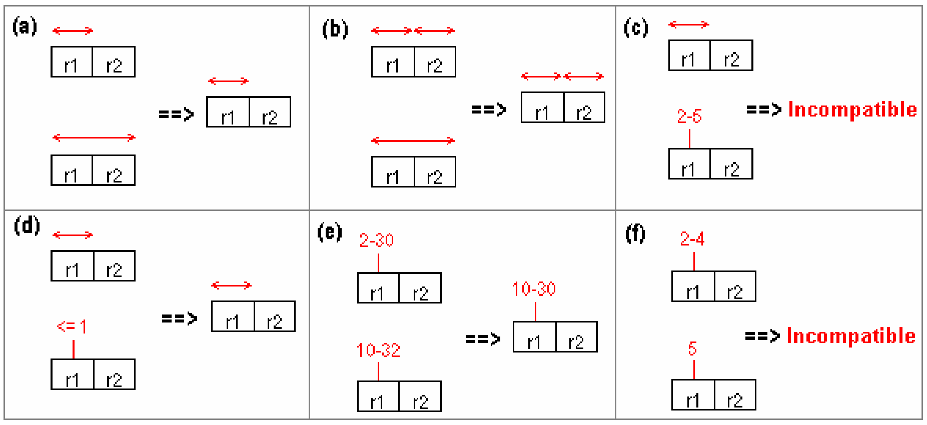

Fig. 6. An example of combining uniqueness and frequency constraints.

Step 2.5: Combining set-comparison constraints

Combining set-comparison constraints across two modules is performed in the following steps:

- Each exclusion constraint that spans more than two singles or sequences of roles (called "multiple" exclusion) is converted into pairs of exclusions[6], such in Fig. 7.

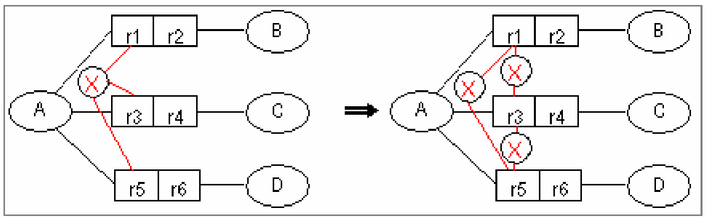

Fig. 4.7. Converting multiple exclusions into pairs of exclusions.

- When combining a subset (or equality) in one module and an exclusion in another, the composition of ($M_1 \oplus M_2$) is considered an **incompatible operation**, because the two constraints contradict each other, and so both roles cannot be satisfied. See fig. 8.

- As equality implies subset (but not vice versa) [H89], when combining a subset in one module and equality in another module, or when combining two subset constraints that are opposite to each other, the combination is always equality. See Fig. 9.

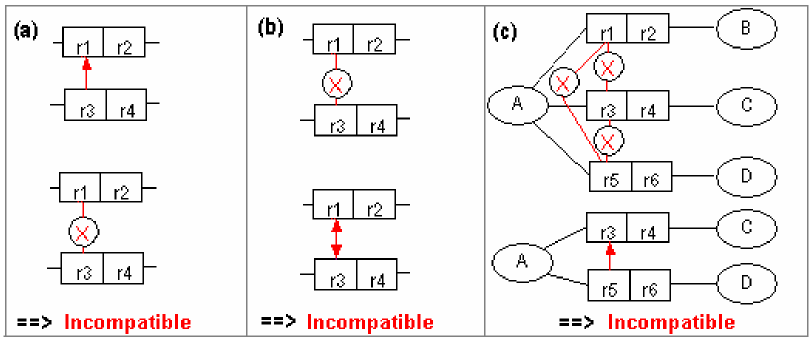

Fig. 8. Combining subset (or equality) with exclusion.

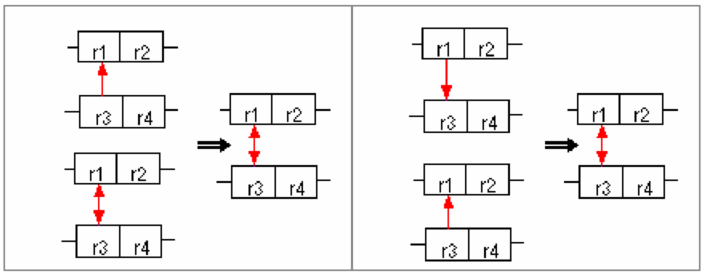

Fig. 9. Combining subset and equality constraints.

[6] This conversion is temporary for reasoning purposes, so it will not appear in the final result of the composition. Notice that "a single exclusion constraint a cross n roles replaces $n(n\text{-}1)/2$ separate exclusion constraints between two roles" [H01].

Step 2.6: Combining subtype constraints (total, exclusive)

When composing two modules, all subtype constraints are included in the composition without any specific combining operation.

Step 2.7: Combining 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 types of ring constraints are supported by ORM: antisymmetric (ans), asymmetric (as), acyclic (ac), irreflexive (ir), intransitive (it), and symmetric (sym) [H01][H99]. The relationships between the six ring constraints are formalized by [H01] using the Eular diagram as in fig. 10. This formalization helps one to visualize the implication and incompatibility between the constraints. For example, one can see that acyclic implies reflexivity, intransitivity implies reflexivity, the combination between antiasymmetric and reflexivity is exactly asymmetric, and acyclic and symmetric are incompatible.

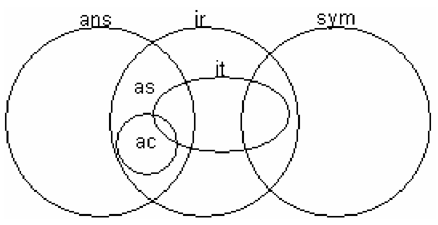

Fig. 10. Relationships between ring constraints [H01].

When composing two modules, ring constraints are combined based on the formalization in fig. 10. Any combination of ring constraints should be compatible, i.e. there is an intersection between their zones in the Eular diagram, e.g. see fig. 11 (a). Otherwise, the composition of ($M_1 \oplus M_2$) is considered an **incompatible operation**, because the combined rings constraints conflict each other, and thus the role cannot be satisfied. See fig. 11 (b).

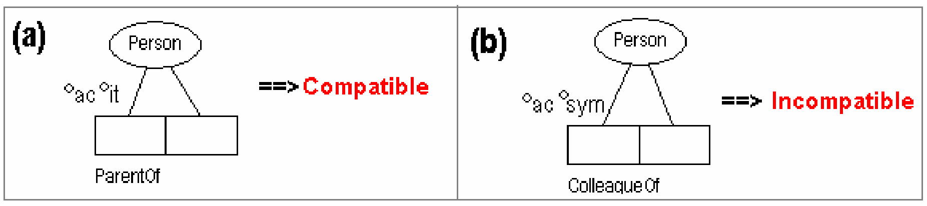

Fig. 11. Examples of compositions ring constraints.

4 Discussion, conclusions and future work

This paper has presented an approach to modularize and automatically compose ORM schemes. This approach is fully implemented in DogmaModeler [J05], which is a software tool for modeling ontologies and business rules using the ORM graphical notation. DogmaModeler enables users to create, compose, add, delete, manage, and browse ORM (modular) schemes. DogmaModeler also implements a library of ORM modular schemes, allowing different metadata standards (e.g. Dublin-Core, LOM, etc.) to be used for describing modules. This approach has been also used in a real-life case study (CCFORM EU project, IST-2001-34908, 5th framework.) for developing modular axiomatizations of costumer complaints knowledge, see [J05][JVM03] for the experience and lessons learned.

Although we assume in our formal framework (in section 2) that the composition is terminated in case of unsatisfiability, it is not necessary for the resultant composition, in our algorithm of composing ORM schemes (in section 3) to be satisfiable, thus our algorithm is called *incomplete*. This is because the general problem of determining consistency for all possible constraint patterns in ORM is undecidable [H97]. A complete semantic tableaux algorithm for deciding the satisfiability of ORM schemes (a research topic by itself) is not a goal of this paper. See our pattern-based approach in [JH05] for reasoning about the satisfiability of ORM schemes.

As an upcoming effort, we plan to map ORM into the ***DLR*** Description Logic [CDLNR98], which is a powerful and decidable fragment of first order logic. In this way, the satisfiability of ORM schemes can be completely verified, and so our algorithm can be called complete. Furthermore, this will allow us to reuse our approach to modularize and compose ***DLR*** knowledge bases.

Acknowledgement. We are in debt to Robert Meersman, Stijn Heymans, Olga De Troyer and Andriy Lisovoy for their comments, discussion, and suggestions on the earlier version of this work.

References

[BBDD97] Briand, L.C., Bunse, C., Daly, J.W. and Differding, C.: An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. In: Empirical Software Engineering, Vol. 2, No. 3. (1997) pp. 291–312.

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

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

[BS03] Borgida A., Serafini L.: Distributed Description Logics: Assimilating Information from Peer Sources. In: Aberer K., March S., and Spaccapietra S., (eds.): Journal on Data Semantics, Vol. 2800. LNCS, Springer, ISBN: 3-540-20407-5. October (2003) pp. 153–184

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

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

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

[H04] Halpin, T.: Business Rule Verbalization. In Doroshenko, A., Halpin, T., Liddle, S., Mayr H. (eds): Information Systems Technology and its Applications, 3rd International Conference (ISTA'2004), LNI 48 GI ISBN 3-88579-377-6, (2004) pp:39-52.

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

[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.

[H99] Halpin, T.: UML data models from an ORM perspective: Part 7. Journal of Conceptual Modeling. InConcept. February (1999)

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

[JDM03] Jarrar M., Demy J., Meersman R.: On Using Conceptual Data Modeling for Ontology Engineering. In: Aberer K., March S., and Spaccapietra S., (eds.): Journal on Data Semantics, Special issue on "Best papers from the ER/ODBASE/COOPIS 2002 Conferences", LNCS Vol. 2800, Springer. ISBN: 3-540-20407-5. October (2003) pp. 185–207

[JH05] Jarrar, M., Heymans, S.: Unsatisfiability Reasoning in ORM Conceptual Schemes. Technical Report, Vrije Universiteit Brussel, 2005.

[JM02b] Jarrar, M., Meersman, R.: Scalability and Knowledge Reusability in Ontology Modeling. Proceedings of the International conference on Infrastructure for e-Business, e-Education, e-Science, and e-Medicine (SSGRR'2002s) (2002)

[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)

[P72] Parnas, D. L.: On the criteria to be used in decomposing system into modules. Communications of the ACM, Vol. 15, No. 12. December (1972) pp. 1053–1058

[S85] Shoval, P.: Essential information structure diagrams and database schema design. Information Systems, 10(4). (1985) pp. 417-423

[SK03] Stuckenschmidt H., Klein M.: Modularization of Ontologies -WonderWeb: Ontology Infrastructure for the Semantic Web. Deliverable 21. WonderWeb Project (IST 2001-33052) (2003)

[SMS+05] Spaccapietra, S., Menken, M., Stuckenschmidt, H., Wache, H., Serafini, L., Tamilin, A., Jarrar, M., Porto, F., Parent, C., Rector, A., Pan, J., D'Aquin, M., Lieber, J., Napoli, A., Stoilos, G., Tzouvaras, V., Stamou, G.: Report on Modularization of Ontologies. Deliverable D2.1.3.1 (WP2.1), KnowledgeWeb project. EU-IST Network of Excellence (NoE) IST-2004-507482 (2005)

[SP94] Spaccapietra, S., Parent, C.: View Integration: A Step Forward in Solving Structural Conflicts. IEEE Transactions on Data and Knowledge Engineering 6(2). (1994)

[SWCH01] Sullivan, k., William, G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. Journal SIGSOFT Software Engineering Notes. Vol. 26, number 5. ACM Press. Issn: 0163-5948. (2001) pp. 99–108

[V83] Vermeir D.: Semantic Hierarchies and Abstraction in Conceptual Schemata. Journal of Information Systems. Vol. 8, No. 2. (1983) pp. 117–124

[VB82] Verheijen, G., van Bekkum, P.: NIAM, aN Information Analysis Method. In: Olle, T.W., Sol, H., Verrijn-Stuart, A. (eds.), IFIP Conference on Comparative Review of Information Systems Methodologies, North-Holland. (1982) pp. 537–590