Ontology Design and Integration with ICOM 3.0 - Tool description and methodology

Pablo R. Fillottrani[1], Enrico Franconi[2], and Sergio Tessaris[2]

[1] Universidad Nacional del Sur, Argentina
[2] Free University of Bozen-Bolzano, Italy

1 Introduction

ICOM is an advanced conceptual modelling tool, which allows the user to design multiple ER or UML class diagrams with inter- and intra-model constraints. Complete logical reasoning is employed by the tool to verify the specification, infer implicit facts, devise stricter constraints, and manifest any inconsistency.

For the ontology creation and maintenance tasks, ICOM interface supports ontology engineers in engineering ontologies that meets clear and measurable quality criteria. Indeed, recently we observe the development of large numbers of ontologies which have, however, usually been developed in an ad hoc manner by domain experts, often with only a limited understanding of the semantics of ontology languages. The result is that many ontologies are of low quality - they make poor use of the languages in which they are written and do not accurately capture the author's rich knowledge of the domain. This problem becomes even more acute as ontologies are maintained and extended over time, often by multiple authors. Poor quality ontologies usually require localised "tuning" in order to achieve the desired results within applications. This leads to further degradation in their overall quality, increases the brittleness of the applications that use them, and makes interoperability and reuse difficult or impossible. To overcome these problems tools are needed which support the design and the development of the basic infrastructure for building, merging, and maintaining ontologies.

The leverage of automated reasoning to support the domain modelling is enabled by a precise semantic definition of all the elements of the class diagrams. The diagrams and inter-model constraints are internally translated into a class- based logic formalism. The same underlying logic enables the use of a view definition language to specify additional constraints, not captured at the diagram level. The conceptual modelling language supported by ICOM can express most of features of the Extended Entity-Relationship data model or the UML class diagrams (we are working on supporting *Object-Role Modelling* [5] as well). Moreover it extends with disjoint and covering constraints and definitions attached to classes and relations by means of view expressions over other classes and relationships in the ontology; as well as inter-ontology mappings, as inclusion and equivalence statements between view expressions involving classes and relationships possibly belonging to different ontologies.

The main purpose of the ICOM project is not to provide to the ontology community a robust tool potentially replacing the many other tools available, and we do not claim that ICOM is currently more usable than any of the existing conceptual modelling tools

for ontology design (such as, for example, [8, 1]). ICOM is meant to be a proof of concept, willing to showcase two main points: (1) the effectiveness of using a class diagram graphical syntax for expressing ontologies, even with complex languages; (2) the emphasis to the use of complex automated reasoning tasks to deduce implied facts, as opposed to mere subsumption (classification) and consistency.

The two above points are novel and in our opinion very important in the context of the existing ontology design tools and methodologies (see next Section). Indeed ICOM proves (point 1) the feasibility and the ease of use of a class diagram graphical syntax for expressing ontologies, even with complex ontology languages, by relying on the notion of views (which roughly correspond to OCL constructs) in order to capture the (typically very few) cases where a larger expressivity than graphical class diagrams is needed.
ICOM is based on a *deduction-complete* notion of reasoning support relative to the class diagram graphical syntax (point 2). Users will see the original ontology *graphically completed* with all the deductions making sense given the provided ontology, and expressed in the graphical class diagram language itself. This includes checking class and relationship consistency, discovering implied class and relationship inter-relations (e.g., subsumption) or cardinality constraints, and in general discovering any implied but originally implicit class diagram graphical construct. Customarily, ontology design tools just provide a support limited to class subsumption and consistency.

ICOM is a fairly mature project, its first release has been published in 2000 (see [7, 4]). The version 3.0 of the ICOM tool is loosely based on the ICOM tool previously released in 2000 as an Entity-Relationship editor (which had around 3,000 registered installations, mostly in academic environments and for teaching purposes in industry), and a demo of a preliminary version was presented few years ago [3]. The foundations of the user-computer interaction have been radically changed according to the experience of the first ICOM and the research in this last decade. The system has been completely re-implemented, using different graphic libraries. The graphical interface has been completely rewritten to improve the usability and intuitiveness of the tool. Interoperability with other tools is a crucial aspect; so, import and export modules have been developed for XMI 2.x and Description Logics based ontology languages via DIG.

The ICOM tool is written in standard Java 5.0, and it is distributed on Linux, Mac, and Windows machines.[3] ICOM communicates via the DIG 1.1 protocol with a description logic server, such as, for example, RACER. ICOM provides an interface for importing and exporting ontologies in UML-XMI class diagrams format.

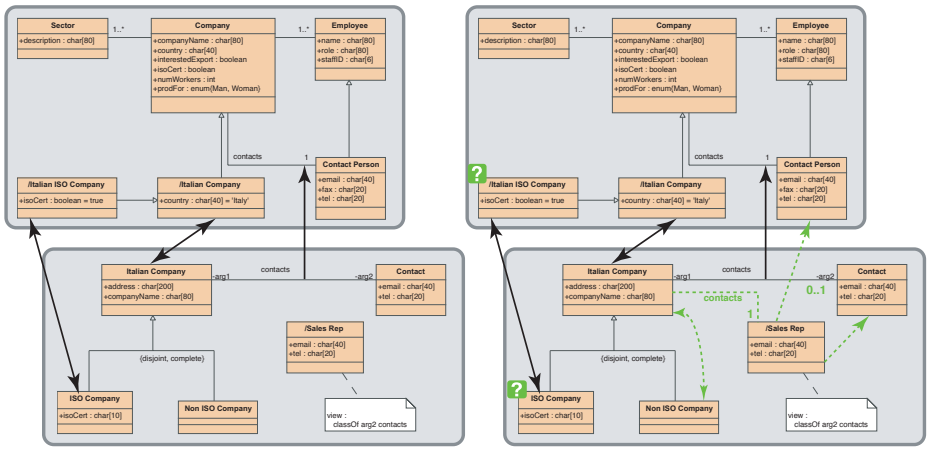
2 Ontology Integration and Views in ICOM

In this section we introduce a scenario of usage of the tool in the context of ontology integration by making use of the view facility of ICOM.

Figure 1a shows two ontologies in the phase to be integrated by the ontology engineer. The top ontology describes concepts where information about Italian ISO certified companies is held; in particular, the information about their contact person is specified.

[3] Available as a free download at `http://www.inf.unibz.it/~franconi/icom/`

The facts described by the diagram state that a company should have at least one employee, and that it should be involved in at least one sector. Among the employees there is the contact person of the company, which should be unique. Moreover, the Italian companies are exactly defined as those companies which are in a country called Italy, while the ISO certified Italian companies are exactly those Italian companies having an ISO certification (specified as a boolean property called `isoCert`). Please note the particular use of the 'slash' "/" operator in front of the completely intensionally defined classes—in the ontology the classes `Italian Company` and `Italian ISO Company` are completely defined by means of the properties specified in the diagram. This is the simplest case of a view defined in the ontology.



(a) Without deductions (b) With deductions

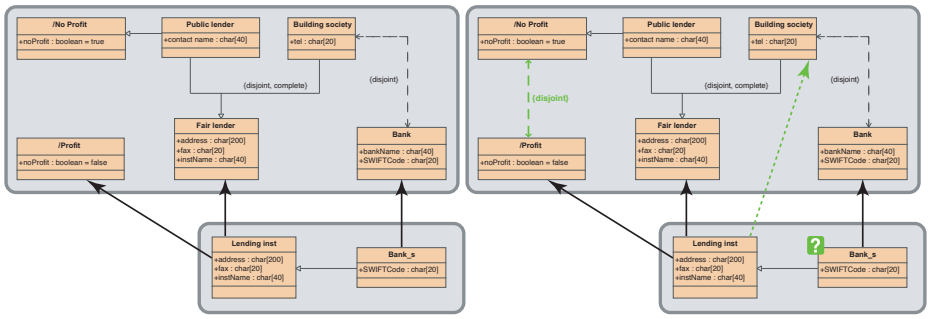
Fig. 1: The first integration scenario.

The lower ontology of Figure 1a describes a slightly different perspective about the same domain. Still, there are Italian companies and their contact persons (but now without any cardinality constraint, and without mentioning that contact persons of companies should be employees), plus the specification that the companies are either ISO certified or not—here the ISO companies are identified by the code of their ISO certification institution. In addition, the ontology includes the view class `Sales Rep`, which is completely defined by means of its attributes together with the view expression stating that sales representative is the range of the `contacts` association. Note that the view definition can be written in any *reasonable* textual ontology language, such as an OCL constraint, or an OWL axiom, or a SQL check constraint, or first order logic sentence. The view definition mechanism is the hook that allows to use the full power of the ontology language—if the user wants. Most of the ontologies will not need to use this hook, and they will be more directly understandable by the engineers. In the case when subtle integration constraints have to be written, views will come in handy, by providing

an expressive language to the engineers in a way which is perfectly integrated with the diagrammatical paradigm proposed here.

Figure 1a includes also the mappings between the two ontologies. You can see that the Italian ISO companies in the top ontology are declared to be the same as the (Italian) ISO companies in the lower ontology, and that the Italian companies in both ontologies are declared to be equivalent as well. Moreover, the `contacts` association in the lower ontology is declared to specialise the homologous association in the top ontology. Inter-ontology mappings are declared by simply drawing directed links between pairs of classes (or pairs of associations) belonging to different ontologies; these can state either equivalence, or containment, or disjointness.

Now, the whole picture seems very reasonable to any ontology engineer; however there are interesting, unexpected, and clarifying consequences that our design tool will automatically draw—still in a diagrammatic fashion. These are shown in Figure 1b.



(a) Without deductions (b) With deductions

Fig. 2: The second integration scenario.

The first consequence relates to the equivalence stated between the two Italian ISO certified company class definitions in the two ontologies. The two classes have a type incompatibility in the attribute `isoCert`: one is declared to be a boolean value, while the other is declared to be a string of ten characters. Indeed, the system deduces that if such an integration has to be taken seriously, then the two classes have to be empty in any possible context, since an object in one context which, say, represents an Italian ISO Company by having an attribute `isoCert` with the value true, can not be at the same time an instance of a class whose `isoCert` is declared to be of an incompatible type (i.e., string). Therefore, such an instance can not exist, and, as a matter of fact, no instances of the two classes can exist at all. This first deduction by the tool (indicated by the question marks on the corner of the two classes) is actually a hint to the designer to actually take care of this *data reconciliation* problem, by, for example, providing local conversion functions between the two attribute types. Please note that the tool also correctly derives the fact that any object which is instance of the Italian Company class (in any of the two contexts/ontologies) should also be an instance of a non ISO

company. In fact, since no Italian ISO companies can exist in the current version of the scenario, any Italian company will necessarily be a non ISO certified company. This is made explicit by the dashed equivalence link added by the tool between the `Italian Company` class and the `Non ISO Company` class in the lower ontology.

If we go on with the analysis of the deductions made by the tool, we see that a stricter cardinality constraint has been deduced: now any Italian company can have at most one contact person (in the sense of the lower ontology)—this is the `[0..1]` cardinality constraint found at the right end of the `contacts` association. The system has deduced this stricter constraint by observing that Italian companies are companies, which have exactly one contact person (in the sense of the top ontology); moreover, each Italian company should have contact persons (in the lower ontology sense) among the contact persons in the top ontology sense. Therefore, no Italian company can have more than one contact person (in the lower ontology sense). The lower bound is not derived since the specialisation of the `contacts` association may not necessarily consider all the Italian companies. So, this is an example of a deduction which is not just an IS-A link or an inconsistent class, which are the only kind of deductions that the most advanced ontology design tools (like, e.g., OILEd, or Protege) are capable of.

Another deduction which can not be done by any other ontology design tool is the one which makes explicit a `contacts` association in the lower ontology between Italian companies and sales representative, plus the now stricter cardinality constraint stating that each Italian company has exactly one sales representative. Please note how powerful this deduction mechanism is: an isolated class is automatically fully put in context, by considering all the possible constraints which may relate it to the other terms of the integrated ontologies. As a matter of fact it can be proved that the design tools derives *all* (and only) the implied constraints representable within the diagrammatic ontology language.

Finally, we note that the tool derives also that sales representative are both contact persons in the top ontology sense and contact persons in the lower ontology sense.

All these deductions may help the ontology engineer in validating the design—if the derived constraints make sense to the engineer; they may help in suggesting changes; or they may show serious but subtle conceptual mistakes. The next case scenario shown in Figure 2a is an example of the latter case.

In this new integration scenario, the top ontology describes fair lenders which are partitioned into public lenders and building societies. Public lenders are no profit companies, and in addition it is stated that banks are not building societies.

In the lower less detailed ontology, we have the generic class of lending institutions which specialises into the bank class. We also assume that actually the lower ontology, in spite of the fact that it uses more generic terms, describes a world which is actually a portion of the world described more accurately by the top ontology.

A very natural integration between the two ontologies is pursued by the ontology engineer: she/he states that banks of the lower ontology are among the banks of the top ontology, and that lending institutions of the lower ontology are fair lenders and profit companies as defined in the top ontology.

The consequences of this integration attempt are immediately drawn by the tool as depicted in Figure 2b. As a first (more or less obvious) deduction we can observe that

the profit and the no profit classes are derived to be disjoint, as expected. However, it turns out—from the big question mark at the corner of the `Bank_s` class—that no banks can exist according to the lower ontology! This is somehow unexpected, since we thought we were playing a rather simple game in this case. Why is this? A quick glance at the attribute types shows that they are perfectly compatible this time. The reason is the following. First of all, we can derive that lending institutions are building society (as pointed out by the tool); in fact, lending institutions are fair lenders, which can be either public lenders or building societies. On the other hand we have that lending institution are profit companies, which are provably disjoint from public lenders. therefore, any lending institution has necessarily to be a building society. At this point, we are closer to the answer to our original question about the inconsistency of the `Bank_s` class. Those lower ontology banks are at the same time a kind of top ontology banks and building societies (by transitivity). However the two latter classes are disjoint, hence no common instance can exist—i.e., no bank can be a lending institution according to this integration system.

Of course, there should be something wrong the way the two ontologies have been integrated, and this calls for a revision of the mappings. The engineer should either omit the mapping stating that lending institutions are necessarily profit companies, or the mapping stating that lending institutions are necessarily fair lenders. In both cases, the outcome will be acceptable by the engineer, and she/he should choose the one that best fits further analysis of the domain that she/he may have done after this unexpected discovery.

3 The Ontology Editor

The Ontology Editor works on *projects*, which may contain one or more UML class diagrams. The diagrams are referred as *models*. Multiple projects can be opened at the same time, but objects cannot be moved across them. Only one project is visible at a time and the editing of each project is independent. The user can switch between different projects using the tabs at the top of the project area. Classes are represented by boxes and n-ary associations by diamonds. Associations may have so-called association classes specifying their attributes. IsA relationships are represented as arrows with a disc in the middle (e.g. see `MobileCall` and `Call`).

The tool does not implement special visual techniques for handling very large ontologies. The tasks that it supports, i.e. authoring of concept description and structuring the ontology, are not aimed at working simultaneously with thousands of concepts. However, a set of functionalities that are very useful in managing such ontologies are available. First, the interface is zoomable, that is, the level of detail and size of the icons that represent the model can be smoothly changed by pressing the right mouse button and dragging left to zoom our or right to zoom in. Also, the window can be panned by pressing the middle button and dragging. This allows the user to focus the attention in a specific region of the ontology. There are also two dedicated buttons for zooming: one will show the complete graph, and the other will zoom in to show the selected elements. Selection works by left-clicking on icons or by left click and drag; also, there is a button for expanding the selection to all connected nodes, which is very useful in

combination with the zoom-to-selection button. Finally, custom automatic layout algorithms for ontologies are under development. These combine known layout algorithms for drawing large graphs, with special conventions used in ontologies, like IsAs hierarchies are drawn top-down and associations are drawn in the middle of related concepts. New metrics to measure the "quality" of ontology graphs were developed for this purpose.

Editing Models Most of the model editing is done in the project panel, where each model in the project is displayed in a separate model panel. In addition, two dialogues are used to elicit additional information about model objects. The attribute domain dialogue allows the domain of attributes to be set. The definition dialogue enables the characterisation of a class or association by means of a view written in the language described in the next Section.

Objects can be created by selecting the appropriate button in the toolbar, or an entry under the Diagram menu, or a contextual menu in the project area. Most object-creating operations require further inputs to complete the operation. Usually, the user is requested to select an existing object in the diagram (e.g. during the creation of an IsA relationship). In this case, the system will highlight only objects in the diagram suitable for the specific operation.

All the objects of the diagram have a name. Upon their creation the system allocates a new fresh name, which can be edited by the user. To improve the identification of the nodes, when icons become smaller because of the zoom level, all the nodes show their name on a tool-tip when the mouse is hovering over them. Names are scoped by the model they belong; e.g. classes with the same name in different models are considered different.

Metadata fields can be associated to every kind of objects. These fields are ignored in the reasoning process.

The creation of a new class adds a new box in the diagram with a new default name. Every class can optionally have attributes. Attributes are added and edited by means of a specific attribute dialogue. Similar to classes, attributes of the same name in different models are considered different. Attributes of the same name within the same model represent the same attribute. For each attribute, a domain should be indicated. There the set of possible domains is not predefined, and the user is allowed to enter an arbitrary name. Unlike the classes and associations, domains have a global context. Therefore, domains of the same name in different models are considered be the same.

Associations are created by default with no roles. N-ary associations can be specified by adding new roles to existing ones. The creation of a new association introduces a corresponding association class, which can be edited as a normal class (e.g. it can have attributes).

Adding new roles to an existing association requires the user to select the association and a class which restricts the domain of the argument of the association corresponding to the role. Similar to class and associations names, role names have a model scope.

For example, assume there are two models `M1` and `M2`, each one with a binary association `lives` having the roles `subject` and `object`. Note that, being association scoped over models, from the global perspective there are two associations `M1:lives` and `M2:lives`. Now, the modelling of the domain requires that `M2:lives` is more

specific (i.e. a subset) of `M1:lives`. Since also role names are scoped over each model, overall there are four different roles. Therefore, the more specific association (`M2:lives`) *inherits* the roles of the general one, ending up being of arity four (namely the roles `M2:object`, `M2:subject` and `M1:object`, `M1:subject`).

Roles denote the connection of a class to an association and are also used to express the cardinality constraints of a class in an association. A role may have two constraints: *totality*, or the minimum cardinality, and *uniqueness* representing the maximum cardinality. In the current version of the system, the numbers expressing cardinality are restricted to be 0 and 1. A minimum cardinality of 1 indicates that all instances of a class must participate in the association at least once (i.e. mandatory constraint). A maximum cardinality of 1 indicates that all instances of a class can only participate once in the association (i.e. functional constraint).

Within a project, equivalence and subset role mappings can be defined between roles in the same or different models. These allow a better characterisation of the relationship between associations across different models. In the former example, `M2:lives` can be set as a binary association by saying that `M1:object` contains `M2:object`, and that `M1:subject` contains `M2:subject`.

The system enables the user to specify inheritance relationships among classes and associations. The relationships can be arbitrary (e.g. cycles are allowed) provided that classes can only inherit from classes, and associations from associations. Formally, the inheritance is expressed by the inclusion (subclass) constraint.

On the diagram, inheritance is specified by means of IsA links (in the diagram indicated by arrows with a circle in the middle) connecting nodes. IsA links can be specified one-to-one, or many-to-one. The latter groups together more than one (association) class and restrict all of them to be a subclass of the link target.

The possibility of grouping more than one descendant, not only provides way of visually organising the layout of the model; but enables the user to specify additional constraints among the (association) classes. In particular, the *covering* and *disjointness* constraints. The first one expresses the fact that the (association) class is equivalent to the union of the specified descendant, while the second constraints the grouped (association) classes to be mutually disjoint.

Note that disjointness among classes is not assumed by default; so, in absence of a specific constraint, (association) classes may overlap.

Inter-Model Axioms Additional constraints among classes and associations can be expressed by means of intra- as well as inter-model axioms. The Ontology Editor provides four types of axioms: Node Definition, Equivalence, Subsumption and Disjointness. As discussed in Section 1 these constraints provide a powerful modelling tool in the context of data integration and ontology mapping.

Each class and association can be fully defined by means of a view expression. The view expression language is more expressive that the diagrammatic definition language, so enables the expert user to add constraints that cannot be expressed by the UML diagram alone.

The adopted view language is based on the DLR description logic. A definition has a global context, meaning it can express inter-model relationships as well as intra-model relationships. The view language includes two syntactic sorts: one for classes and one

for associations. Full boolean operators are allowed, plus a *selection* operator (selecting tuples in an association with a specific class type in some named role argument) and a unary *projection* operator (projecting an association over a named role argument). A generalised projection operator with cardinality restrictions is available as well.

Since a definition can refer to objects in different models, a name-prefix is used in definitions to distinguish objects with the same name but from different models. The name-prefix used is the model's name followed by a colon symbol. For example, `class1` in `Model1` and `class1` in `Model2` would be referred to as `Model1:class1` and `Model2:class1` respectively.

Any two (associations) classes in any model can be related by semantic relationships stating their equivalence, subsumption, or disjointness. Creating one of these relationships requires the user to specify source and target node. The system prevents the creation of a relationship between non-homogeneous nodes by restricting the scope of the second node to be selected.

Exporting and Importing Projects ICOM projects can saved and retrieved in an own XML format, preserving the meaning of all elements including view definitions. It is also possible to import UML class diagrams saved in the XMI format. The tool only recognises the subset of XMI determined by classes, associations, attributes, roles and primitive datatypes defined within an UML model. Functional and mandatory constraints on roles are the only type of imported constraints. Aggregation relationships in the UML model are ignored. We are currently working on exporting projects in XMI files, but this translation would be necessarily carried out with some loss of meaning because, for example, not all view definitions can be expressed in XMI even with attaching OCL expressions to the model elements (e.g. names in OCL expressions have a scope which is local to a given class, rather than global as in ICOM).

4 Automated Reasoning

Although the Ontology Editor can be used as a standalone modelling tool, exploiting its full capabilities requires the coupling of the system with a Description Logic reasoner. Without such an automated reasoning tool the Ontology Editor would be unable to perform *deduction-complete* automated reasoning over the models. As we noted, this includes checking class and relationship consistency, discovering implied class and relationship inter-relations (e.g., subsumption) or cardinality constraints, and in general discovering any implied but originally implicit class diagram graphical construct.

Instead of implementing its own dedicated reasoner, the Ontology Tool can exploit any DIG enabled DL reasoner (see [2]). Being DIG a standardised communication protocol, the user can choose the most suitable DL reasoner (e.g. the one used by other in-house project), or upgrade to the latest version of the preferred reasoner without being forced to upgrade to a different version of the Ontology Editor.

The so called *verification process* can be computationally expensive, so it is activated only on user's request. This process includes the following operations. The selected project is encoded into an appropriate Description Logics knowledge base and shipped to the DIG reasoner. Each class, association in the project is checked for satisfiability (i.e. non-emptiness). For each class, association in the project, its equivalent

peers, and super-classes are determined. For each class-role-association triple, the system calculates the stricter minimum and maximum cardinality constraints. To perform these operations, the system formulates a sequence of queries to be sent to the DIG reasoner. Accordingly to the received answers the Ontology Editor infers properties of the models in the project. To perform these operations, the system formulates a sequence of queries to be sent to the DIG reasoner, which is linear in the number of project elements. Accordingly to the received answers the Ontology Editor infers properties of the models in the project. The algorithm for this inference is quadratic in the number of concepts and roles, and linear in the number of axioms and IsA links. Thus, the tool can reasonable manage projects with several hundreds of elements, calling a current state-of- the-art reasoner.

After the verification process, the system provides the user with a visual account of the deductions by modifying the appearance of the model diagrams in the project. All unsatisfiable objects will appear in red in the model diagrams. An object is unsatisfiable when necessarily describes an empty set of tuples of objects. Additional non explicit deductions will appear in green, to be distinguished from the user specified elements of the diagrams. Semantically equivalent objects are connected with newly inserted equivalent axiom links. Objects discovered to hold an inclusion relationships between them are connected with subsumption axiom links. Cardinality constraints which are stricter than those originally specified. Although the deductions are displayed on the actual diagrams, it is up to the user to decide whether they should be permanently added to the models or discarded. The rational behind this behaviour is that the automated reasoning process may detect unwanted deductions caused by a wrong modelling of the domain. In this case the user should correct the project before any subsequent editing. Another reason is that, in spite of the fact that only the non-trivial deductions are presented, the user is satisfied by the fact that they are implicit without the need of having them explicitly asserted.

The user can discard the deductions and the entire project will be returned to its original state (and any information about unsatisfiability will be discarded). Editing one of the models in the project will also discard the deductions before the editing is carried out. Alternatively, the equivalence, subsumption association, and role cardinality deductions can be added permanently to the project by committing them.

5 Conclusions and Future Works

In this paper we presented ICOM, an advanced conceptual modelling tool grounded on more than ten years of research on the use of automated reasoning to support the development and integration of ontologies. ICOM employes a diagrammatic based language to represent most of the constructs used in ontology design; although it enables the use of non graphical ontology languages, experience with users demonstrates that the design of the diagrammatic language is sufficiently expressive to describe rich domains. Moreover, deductions are expressed within the same diagrammatic language, providing a uniform view over design and analysis of models.

By means of use cases we demonstrated the importance of exploiting basic reasoning tasks (such as subsumption) in order to provide richer information on ontologies.

This is a crucial step towards guaranteeing the quality of the ontologies designed using a tool like ICOM.

The research and development of ICOM continues on two main tracks: from one side we are improving the modelling workflow by considering alternative modelling languages and reasoning services, while on the other hand we are enhancing the user experience by improving the graphical user interface and the interoperability.

We are currently considering the adoption of modelling features from ORM [5] conceptual modelling methodology and representation. Its adoption would have the advantage of leveraging the vast research which has been carried on supporting the user in the modelling tasks, including the integration of natural language generation. The use of ORM modelling style would require also a redesign of the reasoning tasks in order to align the inferences to the new graphical representation.

On the interface we are improving the automatic layout algorithms and working on the support of undo actions. We also plan to include a role browser tab to show the role hierarchy in the same style of the class browser. Moreover, we are improving the interoperability with other tools by tackling the import and export compatibility with XMI and OWL.

References

[1] S. Bechhofer, I. Horrocks, C. Goble, and G. Stevens. Oiled: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001*, pages 396–408. Springer-Verlag, 2001.

[2] S. Bechhofer, R. Möller, and P. Crowther. The dig description logic interface. In *Proceedings of DL 2003*, volume 81 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[3] P. Fillottrani, E. Franconi, and S. Tessaris. The new icom ontology editor. In *Proceedings of DL 2006*, volume 189 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[4] E. Franconi and G. Ng. The i.com tool for intelligent conceptual modeling. In *Proceedings of the 7th International Workshop on Knowledge Representation meets Databases (KRDB 2000)*, volume 29 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2000.

[5] T.A. Halpin, A.J. Morgan, and T. Morgan. *Information modeling and relational databases*. Morgan Kaufmann series in data management systems. Elsevier/Morgan Kaufman Publishers, 2008.

[6] M. Horridge, S. Bechhofer, and O. Noppens. Igniting the owl 1.1 touch paper: The owl api. In *Proceedings of OWLED 2007*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[7] M. Jarke, C. Quix, D. Calvanese, M. Lenzerini, E. Franconi, S. Ligoudistianos, P. Vassiliadis, and Y. Vassiliou. Concept based design of data warehouses: The dwq demonstrators. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, page 591. ACM, 2000.

[8] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The protégé owl plugin: An open development environment for semantic web applications. In *Proceedings of ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*. Springer, 2004.

[9] T. Liebig, M. Luther, O. Noppens, M. Rodriguez, D. Calvanese, M. Wessel, M. Horridge, S. Bechhofer, D. Tsarkov, and E. Sirin. Owllink: Dig for owl 2. In *Proceedings of OWLED 2008*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[10] C. Lutz, F. Baader, E. Franconi, D. Lembo, R. Möller, R. Rosati, U. Sattler, B. Suntisrivaraporn, and S. Tessaris. Reasoning support for ontology design. In *Proceedings of OWLED 2006*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.