Representation and Management of Reified Relationships in Protégé

- Extended Abstract -

Tania Tudorache <tania.tudorache@daimlerchrysler.com>
Knowledge Based Engineering Department, DaimlerChrysler AG, Berlin

Abstract

For the engineering domain, the basic knowledge representation concepts and methods are often not enough, richer representations are needed in order to capture the complexity of a technical system. One requirement is the ability to represent and support relationships between objects that can hold more information than simple slots, that are addressable as objects and can be used in reasoning. This paper presents the current approach taken in Protégé for working with reified relations and proposes their extensions with other common types of relationships. The paper also investigates what kind of tool extensions and automations are needed in order to support reification.

1. Introduction and Motivation

Engineering domains tend to be quite challenging with respect to knowledge modeling requirements. Part-whole relations, topological and functional models, different kinds of physical constraints, different views have to be taken into account. Relevant information describes complex dependencies and the way objects are related to each other. Meta-data like author, creation date, version, documentation, but also the type of a correlation (causal, functional, etc.), priorities and other attributes are needed for capturing all aspects of the interconnection between objects in a technical system. Therefore, the relation representation has to be updated, a more powerful mechanism like reification that allows the addition of attributes to a relationship is needed.

An ontology editor should support the editing, management and visualization of reified relationships in an automatic and transparent way for the user, which may expect the same services for them like the ones for "simple" relations represented as slots. Also, an API for the programmatic access to the reified relationships should be defined as an extension to the current API.

The full paper shows how the reified relationships are currently modeled in Protégé and proposes their extension with other types of relations. It also addresses the problems that appear when using reified relationships, shows what automation and consistency checking support is needed, and also suggests modeling and implementation solutions to these issues.

The next section provides a background about modeling relationships, the definition and what types of relationships are identified. The instruments for representing relationships in a frame-based system are also analyzed and small examples are given. Section 3 enumerates and explains the requirements to an ontology editor for the support of reified relationships. Section 4 analyzes the current way of representing a reified binary relationship in Protégé and based on it proposes the modeling for other types of relationships which will be discussed in more detail in the full paper.

2. Background

This section gives a brief overview about representing relationships, presents different categorizations of relation types, and describes the representational instruments in a frame-based system. The detailed description follows in the full paper.

2.1. What Is a Relation?

In OKBC, a relation represents the dependency between concepts in a domain. The Frame Ontology identifies several types of relationships together with their corresponding axioms. We can define several criteria for categorizing relationships: cardinality (one-to-one, one-to-many, many-to-many), ordering (total-order, partial-order, no-order), symmetry (symmetric, asymmetric, antisymmetric), transitivity (transitive, intransitive, weak-transitive), reflexivity (reflexive, irreflexive), etc.

For the scope of this paper, only some types of relationships will be discussed and modeling schemas in Protégé will be proposed in section 4.

2.2. Representation Instruments for Defining Relations in a Frame-based System

In a frame-based system, slots are appropriate for the representation of binary relationships, as it results from the OKBC definition of a slot: "Formally, a slot is a binary relation, and each value V of an own slot S of a frame F represents the assertion that the relation S holds for the entity represented by F and the entity represented by V (i.e., (S F V) in KIF notation)."
Other supporting mechanisms for the representation of relations in a frame-based system are:

- *Slot overriding at class* - allows the customization of template slot facets at a class.
- *Inheritance of template slots from class to subclass* - the facet values of the slots at class will be inherited, and they can be overridden in the subclasses.
- *Hierarchies of slots – subslots* – allow the user to specify restrictions on each member of a relationship. If a relation is defined as a slot of type Instance at a class, then by the means of subslots, individual restrictions (e.g. cardinality) can be imposed on each allowed class.

The normal way of modeling a relationship between class *A* and *B* in Protégé is by attaching a slot of type Instance and allowed values *B*, to class *A*. If the user needs to make statements about this relationship, then a reified relation has to be used. Reification is defined as "regarding something abstract as a physical thing", "thing-ifying", or turning something as complex as a relationship into an object to be addressed and exchanged. By reifying a relationship, a relation object is being used in the representation. One level of indirection is being introduced, but the relationship can now be addressed as an object, can hold attributes, and it can be used in reasoning. The representation of reified relationships is discussed in section 4.

3. What is Needed in Protégé for the Support of Reified Relationships?

The same mechanisms and operations as for the normal slots should be defined for the reified relationships in Protégé. This means that a standard modeling of the reified relationships is needed, on which all the automations and implementations can be based. The full paper will suggest reified representations for the most common types of relationships.

1. All the reified relation classes should be subclass of a built-in class, for example *:RELATION* like it is now already available. All the other types of reified relationships (directed, bi-directional, transitive, etc.) should be subclasses of this built-in class. In this way, the identification of the relation classes is straightforward.

2. Template models for certain types of reified relationships should be defined, for example by enforcing some special slot names, constraints, etc., like it is done now in the built-in class *:DIRECTED-BINARY-RELATION* with the slots *:FROM* and *:TO*.

3. The inheritance and facet overrides of reified relationships are already supported, when taxonomies of relationship classes are used. So, if class *A* and *B* are connected through the reified class *RelAB*, and *A1* and *B1* are subclasses of *A*, respectively *B*, then *A1* and *B1* can be related trough a subclass of *RelAB*. In this way, the facets constraints, like the allowed classes for the *:TO* and *:FROM* slots, are inherited to the relationship subclass.

4. In a similar manner to the inverse-slot mechanism, the inverse of a reified relationship should be supported.

5. When creating visually or through the API a reified relationship between objects, the relationship class or instance should be created automatically. All the needed slots of the relationship should be filled in automatically.

6. The automatic management of the deletion of a reified relationship. If an instance *A* involved in a reified relationship is being deleted, than the relationship instance should also be deleted automatically. Otherwise, the knowledge base will contain a lot of incomplete relationship instances. The management and consistency check of the reified relationships, can be implemented using the listeners mechanism in Java for a runtime check, or at a given time, some procedures, or rules can be run to remove the incomplete relationships.

7. The query plugins should be able to handle the reified relationships and should be able to "jump" over the relation object. This means that the syntax of the query languages has to be extended, to allow the user to ask for the related object, or the relationship object.

The visualization plugins should also support this kind of relationships. There are already means in Protégé to visualize the reified relations. One of them is using the *InstanceTable* and *InstanceRow* widgets, which can be configured in such a way that a user sees only the related object to an instance. In the case of the directed binary relation, the user can only see the *:TO* slot of the relationship instance. But, the editing and inserting of a new relationship can not be as easily done, since the user has to create himself the relationship instance and fill in correctly the relationship own slots.

4. Representation of a reified relationship in Protégé

Reifying a relationship is making an object out of it, in this way it becomes addressable, attributable, and can be used in reasoning. In a frame-based environment, a reified relationship is represented as an object. The definition of the relationship follows at the class level. For actually binding instances between them, the instantiation of the relation class has to follow, in this way obtaining an instance of a relationship. In the following, the architecture of a reified relationship is given starting from a simple example – the directed binary relationship, that is already available as a built-in class in Protégé. (Fig. 1)

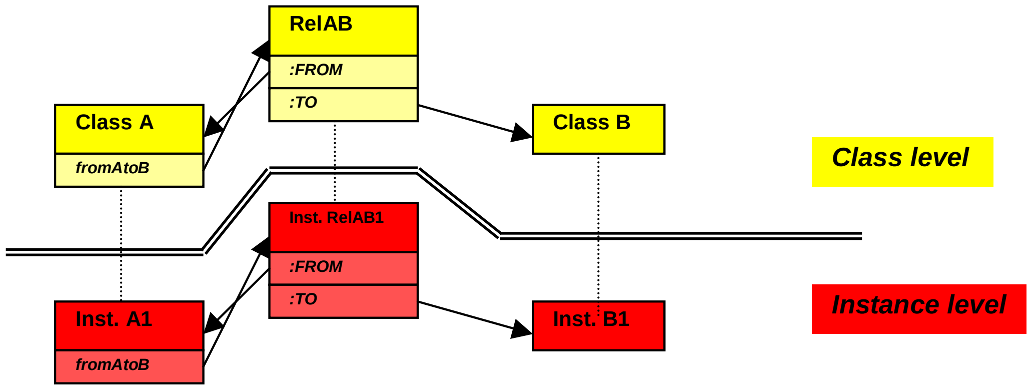

Fig. 1. Representation of a reified relationship. Definition at the class level, usage at instance level.

The *:DIRECTED-BINARY-RELATION* class contains two predefined template slots, the *:TO* and *:FROM* slot. Figure 1 shows how to build a reified relationship between class A and class B through the relation named *fromAtoB*. The template slots are of type Instance. The allowed class for the *:FROM* slot is *A*, for the *:TO* slot is *B*, and for the *fromAtoB* slot, the allowed class is *RelAB*. Attributes that describe the relationship between *A* and *B*, can be added as template slots to class *RelAB*.

The directed binary relationship has been used here as an example, because it is easy to understand, the relation class is built-in in Protégé and it is already supported by some visualization plugins. In a similar way, other types of reified relationships can be defined. The differences come from the consistency conditions that need to be satisfied by the relationship structure (*A*, *B* and *RelAB*) in order to be valid, and the automations needed to build such a relationship.

The full paper will discuss the modeling of other common types of reified relationships: an improved directed binary relation using inverse slots, where the *:FROM* slot will be inferred; the modeling of the bi-directional, inverse relationship which is analog to the inverse-slot mechanism, but applied to the reified relationships; the representation of multidirectional symmetric relationships and the representation of components connections through ports.

A challenge when working with reified relationships is the identification of the elements that belong to the “internal” representation of the relation, that are needed for the interpretation of the relationship. This can be solved by suggesting some standard modeling for the most common types of relationships, and by using some naming conventions (e.g. *:FROM* and *:TO* for the directed binary relation). The full paper will propose another possible solution by using special types of slots (meta-slots) to represent the internal structural elements of a relation object.

5. Conclusions

The reified relationships are an essential representation mechanism for the engineering domain, where the dependencies between the objects are very complex. Therefore, some standard modeling “recipes” for these relationship types are needed together with a corresponding tool support. The full paper analyses the most common types of reified relationships and suggests modeling solutions in Protégé. As future steps, the extension of the current Protégé API for the programmatic access to the reified relationships should also be defined.