A Case Study in Using Protégé-2000 as a Tool for CommonKADS

Guus Schreiber[1], Monica Crubézy[2], and Mark Musen[2]

[1] University of Amsterdam, Social Science Informatics
Roetersstraat 15, NL-1018 WB Amsterdam, The Netherlands
Email: schreiber@swi.psy.uva.nl
[2] Stanford University, School of Medicine, Stanford Medical Informatics
251 Campus Drive, MSOB X-215, Stanford, CA 94305-5479, USA
Email: {crubezy,musen}@smi.stanford.edu

Abstract. This article describes a case study in which Protégé-2000 was used to build a tool for constructing CommonKADS knowledge models. The case study tries to capitalize on the strong points of both approaches in the tool-support and modelling areas. We specify the CommonKADS knowledge model as an ontology in the Protégé specification formalism, and define a number of visualizations for the resulting types. The study shows that this type of usage of Protégé-2000 as a "metaCASE" tool is to a large extent feasible. In particular, the flexible class/instance distinction in Protégé is a feature that is needed for undertaking such a metamodelling exercise. The case study revealed a number of problems, such as the representation of rule types. The study also led to a set of f new tool requirements, such as extended expressivity of the Protégé forms.

1 Introduction and Approach

Knowledge engineering has matured and KE techniques are used increasingly not just for knowledge-system development but also for knowledge analysis and structuring in knowledge management in general. However, the availability of adequate tool support is crucial for a wider adoption of these techniques.

In this paper we describe a case study in which we used the Protégé-2000 tool to build a "knowledge-model editor" for CommonKADS. This editor should support analysts and (to a limited extent) domain specialists in modelling a knowledge-intensive task using the CommonKADS knowledge-modeling framework [11]. The aim was to create an editor that adheres as much as possible to the distinctions made in the CommonKADS knowledge model.

This case study tries to capitalize on the strong points of both approaches. Traditionally, Protégé has put an emphasis on providing configurable and usable knowledge-engineering tools [10, 15]. In CommonKADS the main focus of attention has always been on the modeling side, more or less assuming that support tools would become available in due time. This case study was triggered by the observation that the Protégé-2000 tool is capable of supporting metamodelling. Another interesting feature of Protégé-2000 is the possibility to save the results as RDF [8] (see further). This makes it easier to exchange model data and prevents the common problems experienced by CASE tool

users of getting things out of a tool (the "only input" problem), due to the proprietary formats provided by tool vendors. Finally, the authors thought it would be a sign of maturity of the knowledge-engineering field if two leading methodologies could be linked in this fashion.

1.1 Tool requirements

In this case study we look at the usability of Protégé-2000 from the perspective of three different types of users:

1. The *tool builder*: the person constructing the knowledge-model editor.
2. The *knowledge engineer*: the person that uses the editor to create and maintain a knowledge model. Also, the knowledge engineer should be able to define a domain-knowledge editor to be used by the domain specialist.
3. The *domain specialist*: the person editing and updating the actual domain knowledge of the application.

These three types of users have different requirements. For the tool builder the main requirement is expressivity: can she define the required modelling constructs and visualizations without losing information or clarity? The question whether the interface is easy to use is less important for this type of user, as a high level of expertise in this area is required anyway.

The knowledge engineer needs a tool that enables a convenient and consistent environment for defining a knowledge model. For her it is important that there be no specification redundancy and that consistency and completeness checks for model verification be available. Also, she should be able to include and adapt predefined model parts, such as the catalog of CommonKADS task templates.

The domain specialist will especially be interested in a simple and intuitive interface for updating knowledge bases. Early Protégé research has shown that this requires the use of a domain-specific vocabulary in the user interface [9].

Of course, these requirements are related as the tool created by the tool builder has a strong influence on the functionality offered to the knowledge engineer. The same holds for the domain specialist, who has to use the knowledge-elicitation interface defined by the knowledge engineer. Still, we view these three users as useful perspectives for evaluating the usability of Protégé-2000.

1.2 Related work

Over the years, a number of tools have been developed for knowledge modelling with CommonKADS. An early example is the Shelley workbench [1]. PC-PACK[1] is a contemporary tool with similar aims. PC-PACK focuses on the early phases of knowledge acquisition. It provides functionality for annotating and structuring expertise data such as domain texts, interviews, and self reports. Examples of the use of PC-PACK can be found in various publications [11, 14]. Although PC-PACK offers some support for knowledge-model specification through the GDM grammar [16], the tool can best be

[1] http://www.epistemics.co.uk/products/pcpack/

seen as complementary to a knowledge model editor. The main area in which functionality overlaps concerns the interface for the domain specialist.

WebCokace[2] is an interactive web editor for CommonKADS knowledge models [4]. It uses the CommonKADS CML language [12] to represent knowledge models. The tool is targeted at the knowledge engineer and provides facilities for reusing model elements from catalogs, such as existing task templates and ontologies.

KADS-22[3] is also targeted at the knowledge engineer and supports the knowledge-model editing using the CML version used in the CommonKADS textbook [11]. The tool includes graphical editors and can produce "pretty-prints" of the knowledge model, for example in HTML format.

CASE tools also need to be compared with "low-level" tools, in particular dedicated drawing tools. A CASE tool needs to have a marked advantage in functionality for users to prefer it above such baseline tools. A good example of a baseline tool is ModelDraw[4], developed by Wielemaker. In an evaluation study one would typically want to compare a "heavy" CASE tool with such a light-weight drawing tool.

1.3 Paper overview

In Sec. 2 we briefly summarize the main features of Protégé-2000 and of CommonKADS. Sec. 3 reports on the tool construction process and shows examples of usage of the tool. In Sec. 4 we discuss the experiences gathered during this case study, taking also the different user perspectives into account. Throughout this paper we use examples from an assessment application. This application is concerned with the problem of assessing whether people who applied for a certain (rental) residence conform to the criteria set out for this residence. A full knowledge model of this application can be found in the CommonKADS textbook [11, Ch. 10]. The code of the tool including the example can be downloaded from the Protégé-2000 website.[5]

2 Baseline

2.1 Protégé-2000

Protégé-2000 is the latest incarnation of the series of tools developed for many years by researchers at SMI to provide efficient support in knowledge modeling and knowledge acquisition [7]. Protégé-2000 is platform-independent and offers a component-based architecture, which is extensible through its API. In the rest of this article we use the shorthand "Protégé" to refer to Protégé-2000.

Frame-based knowledge model Protégé is a frame-based environment for knowledge-based system development. Its knowledge model has been re-factored to meet the requirements of the recent OKBC standard [3]. An ontology in Protégé consists of classes, slots, facets and axioms [5]:

[2] http://www-sop.inria.fr/acacia/Cokace/index-eng.html
[3] http://www.swi.psy.uva.nl/projects/kads22/index.html
[4] http://www.commonkads.uva.nl/, see the tools section
[5] http://smi.stanford.edu/projects/protege/

– Class frames specify domain concepts and are organized in a subsumption hierarchy, that allows for multiple inheritance. Classes are templates for individual instance frames.
– Slots are special frames that can be attached to classes to define their attributes, with specific value type restrictions. Own slots define intrinsic properties of class or individual instance frames that do not get propagated either by inheritance or instantiation. Template slots are attached to class frames to define attributes of their instances, which in turn define specific values for slots. Slots in Protégé are first-class objects. They can be specified both globally for the ontology and locally as attached to classes, where their properties are overridden. Each slot is an instance of a metaslot class that defines its properties.
– Facets are properties of slots, which specify constraints on their allowed values. Examples are the cardinality of a slot value, its type (primitive, such as string or integer, or complex, such as instance of a class), range and default values, etc.
– Axioms are additional constraints that can be defined on frames, for example to link the values of a group of template slots attached to a class. As a very recent addition to Protégé, a constraint language enables developers to represent constraints throughout an ontology as sentences expressed in KIF-based [6] predicate logic. Protégé defines a set of built-in predicates and functions that can be used to express constraints. Protégé also provides functionality to evaluate the constraints and check that the individual instances in a knowledge base conform to those constraints. Examples of constraints and constraint evaluation are given in Sec. 3.3.

Configurable forms for knowledge acquisition Protégé perpetuates the support for structured and customizable knowledge entry that has always been fundamental to the Protégé tools. It achieves that goal by providing a configurable user interface for all steps in the process of modeling and acquiring domain- and task-specific knowledge. The graphical user interface of Protégé allows users to define and visualize classes and their slots, to customize a corresponding set of forms for acquiring instances of the classes, and to acquire instances themselves.

The central metaphor for knowledge acquisition in Protégé is the notion of a *form* composed of a set of graphical entry fields ("widgets"). A form is attached to a class to display and acquire its instances. Specific widgets facilitate and locally verify the entry of slot values on instances. Based on the specification of the classes in an ontology, such as the value-type restrictions on their template slots, Protégé automatically generates a form for each class, with default layout and content. Protégé comes along with a set of built-in widgets for the acquisition of slot values, such as a text field for string slots, a pull-down menu for enumerated symbolic slots, a list of instances for multiple instance slots. Users can customize the generated forms to meet domain-specific requirements on knowledge entry and checking. They can rearrange the layout and configure the widget components on the forms, or provide their own plugable widget.

Another kind of knowledge acquisition metaphor that Protégé offers is the notion of a diagram, which provides a means for the synthetic display and acquisition of complex structures defined by a set of related classes in the ontology. Protégé's support for diagrams comes with a special-purpose ontology of metaclasses and classes representing diagrammatic components, that the user extends to create domain-specific diagram

templates. The user defines the nodes of the diagram, that refer to domain classes in the ontology and the types of connectors that can link nodes together. Forms are generated for each diagram construct, that not only take care of graphical layout and navigation but also handle consistency and partial automatic filling of the instances and slot values being acquired through this metaphor. The user is also able to fill-in additional details of instances by zooming into the nodes and connectors.

Flexible class/instance distinction Beyond its OKBC-compliant knowledge model, Protégé now also provides a flexible class/instance distinction based on the notion of metaclasses. A metaclass is a class whose instances are classes themselves. Thus, metaclasses are templates to create new classes and slots in an ontology. They define template slots that are propagated to their instance classes as own slots. Examples are the role and documentation own slots on standard classes.

The metaclass mechanism is described in detail elsewhere [5]. The mechanism implements the internal structure of the Protégé knowledge model itself with a set of built-in metaclasses for classes, slots and facets. The metaclass mechanism also enables developers to customize the underlying knowledge model of their ontology, by defining their own domain- or task-specific metaclasses. Forms for the metaclasses can be customized and instances (new classes and slots in the ontology) can be acquired, similarly to traditional classes and instances. Therefore, this approach extends the scope of modeling possibilities to metamodelling: Developers can use Protégé as an editor for knowledge representation systems with different knowledge models. This case study presents an example of this.

Support for Resource Description Framework Protégé also enables developers to create persistence layer components to import knowledge bases from (and then export to) external storage formats such as a DBMS. This possibility can be combined to the use of the metaclass architecture to redefine a specific knowledge model for a given representation format. This way, Protégé was recently adapted to support the creation and editing of RDF Schema ontologies and the acquisition of RDF instance data [5]. RDF and RDF Schema are the current recommendations from the World-Wide Web Consortium for defining semantic metadata describing Web resources.[6]

The case study that we describe in this paper can be seen as using Protégé to build a specific editor for CommonKADS knowledge bases. As we show in the next section, we made heavily use of the metamodelling constructs offered by Protégé to define the CommonKADS knowledge model.

2.2 CommonKADS

Details about the CommonKADS approach can be found in the recent CommonKADS textbook [11]. CommonKADS is centered around a so-called “model suite”, which takes various different perspectives on a knowledge-intensive task. The central model

[6] http://www.w3c.org/RDF/

is the knowledge model. The use of CommonKADS in this case study is limited to this model. A synopsis of the main constructs in a knowledge model can be found in Table. 1.

Category	Construct	Description
Task knowledge	*task*	a problem statement of what needs to be achieved; specifies also input and output
	task method	specifies a way to achieve a task by decomposing it into subtasks, inferences and transfer functions; also defines a control regimen over the decomposition
Inference knowledge	*inference*	a primitive reasoning function that uses a part of the domain knowledge to achieve a basic problem-solving step
	dynamic role	input or output of an inference; signifies a place holder and an abstract name for domain objects "playing" the role
	static role	the static knowledge used by an inference, also defined as a placeholder for domain objects (e.g., a rule set)
	transfer function	used to denote a primitive function needed to that interact with the outside world
Domain knowledge	*domain schema*	a set of domain-type definitions; a domain schema can be imported into other schemata
	concept	a group of "things"with share features; cf. "object class" or "entity"
	relation	describes a set of tuples that relate "things"to each other; cf. "association", ER-type relationship
	rule type	models expressions about concepts/relations in an antecedent/consequent form
	knowledge base	contains a set of domain-type instances (usually rule instances) that can be used as static knowledge by one or more inferences

Table 1. Constructs in the CommonKADS knowledge model

Most constructs are well known from previous CommonKADS publications, e.g. [13]. A relatively recent addition is the notion of **rule type** in the domain knowledge. A rule type is used to model a set of rules that share a similar structure. For example, in the assessment application we distinguish three rule types: (1) *abstraction rules* that are used to abstract case values (e.g., the age category of an applicant can be derived from the age), (2) *requirements* describing the way in which case values determine truth or falsehood of an assessment criterion such as RentFitsIncome, and (3) *decision rules* that link a boolean combination of criterion values to a particular decision (eligible or not).

Although intuitively the notion of rule type can be easily understood, from a formal point of view it is in fact quite complex. We will see later on that the rule type was the "hardest nut to crack" when constructing the tool.

3 Constructing the Protégé-CK Tool

3.1 Architectural considerations

In order to build a tool for specifying a CommonKADS knowledge model we need to define the knowledge-model constructs as classes and slots in Protégé. For example, a **task** can be represented as a class with slots pointing to the input and output roles, the method that realizes it, etc. Actual tasks will then be instances of **task**. The same can be done for all task and inference constructs listed in Table. 1, i.e., inferences, roles, methods and transfer functions.

However, this approach presents a problem when we come to the domain types. We have three abstraction levels here which we like to represent:

1. Domain-modelling constructs: **concept**, **relation**, **rule type**.
2. Domain-specific types: concept applicant, rule type abstraction-rule.
3. Domain-specific instances: a particular applicant or a particular abstraction rule.

This problem is typical of metamodelling in general. What seems to be an instance from one level, behaves as a class at a lower level. To handle this we can make use of the flexible class/instance distinction made by Protégé. We can define a class as being an instance of a custom metaclass. The resulting class can both be used as an instance (filling in slot values for the slots defined on the metaclass) and as a class (defining template slots for its own instances as well as constraints that should hold). Note that this use of metaclasses is different from approaches in which a metaclass is nothing more than an abstract superclass.

By way of the metaclass architecture we were able to model the three-level approach of the CommonKADS knowledge model. For example, we defined a domain-type metaclass as a template for all domain type classes with an additional template slot rolesPlayed to denote the knowledge role of its instances in the domain. We represented domain-modeling constructs as instance classes of DomainTypeMetaClass, with a root class DomainType. We then modeled domain-specific types as subclasses of the domain modeling classes (thus also instances of DomainTypeMetaClass) with specific values for their rolesPlayed slot. Finally, domain-specific individuals can be acquired as instances of the domain-specific classes. In Sec. 3.3 this approach is illustrated in Fig. 4, where we see its usage in defining the CommonKADS domain-schema constructs.

3.2 Task and inference knowledge

As outlined above, the task- and inference-knowledge constructs were modeled as Protégé classes. An example class definition is shown in Fig. 1 for a task method. The slots correspond to the information that needs to be specified for a task method in the CommonKADS knowledge-modeling language [11, Appendix].

In CommonKADS task and inference knowledge is partially specified through two special-purpose diagrams:

1. The *inference structure* is a diagrammatic representation of inference knowledge, mainly showing how knowledge roles are used as input and output of inferences and transfer functions.

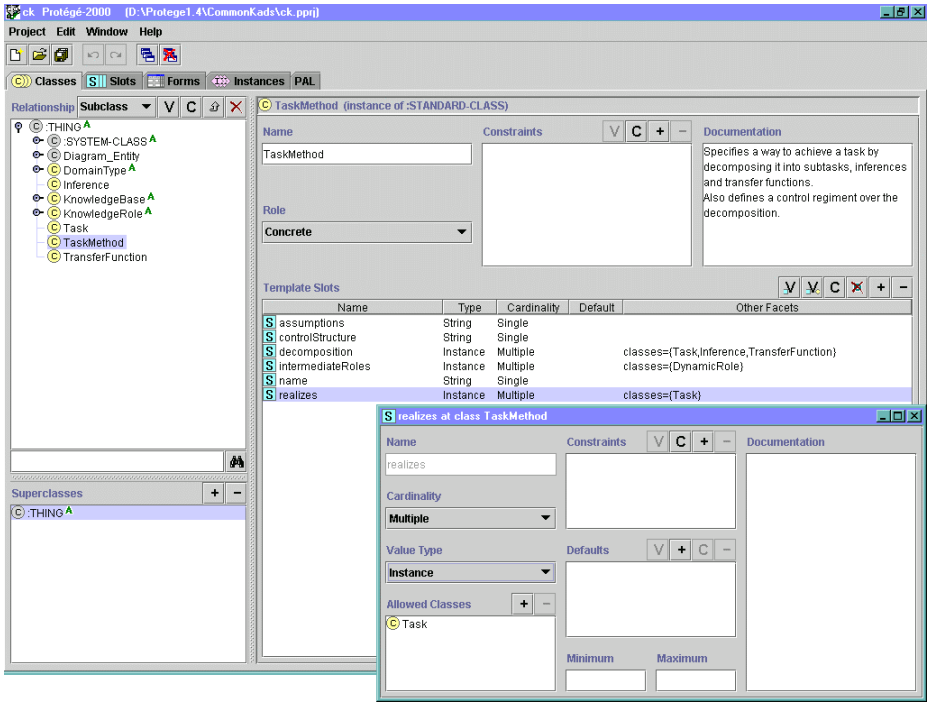

Fig. 1. Specification of the class TaskMethod. At the left, part of the hierarchy of classes is shown. TaskMethod is a root class (i.e., a subclass of :THING, see lower-left). At the right, the class definition is shown. The own slot role indicates whether the class is concrete (can have direct instances) or abstract (no direct instances). Abstract classes are marked with an "A" in the class hierarchy. In the lower-right area the template slots of TaskMethod instances are defined. Each slot has a value range, a cardinality constraint (single or multiple), and a possible value restriction (e.g. the slot realizes can only be filled by instances of the class Task, as shown in the pop-up slot definition window)

2. The *task-decomposition diagram* shows in a tree-like fashion the (recursive) decomposition of tasks through task methods into subtasks, inferences, and transfer functions.

For both the task and inference diagrams, we configured a specialized diagrammatic class in our ontology, as well as an associated form, along the lines described in Sec. 2.1.[7] We defined the InferenceStructureDiagram class with nodes (inferenceStructureNodes) to be instances of the TransferFunction, KnowledgeRole and Inference classes (and subclasses) and connectors to be instances of special-purpose connector classes. For example, the HasInputRole connector links a DynamicRole (subclass of KnowledgeRole)

[7] For details, see the Protege tutorial on the use of the "diagram widget" at http://smi-web.stanford.edu/projects/protege/protege-2000/doc/tutorial/diagrams/index.html

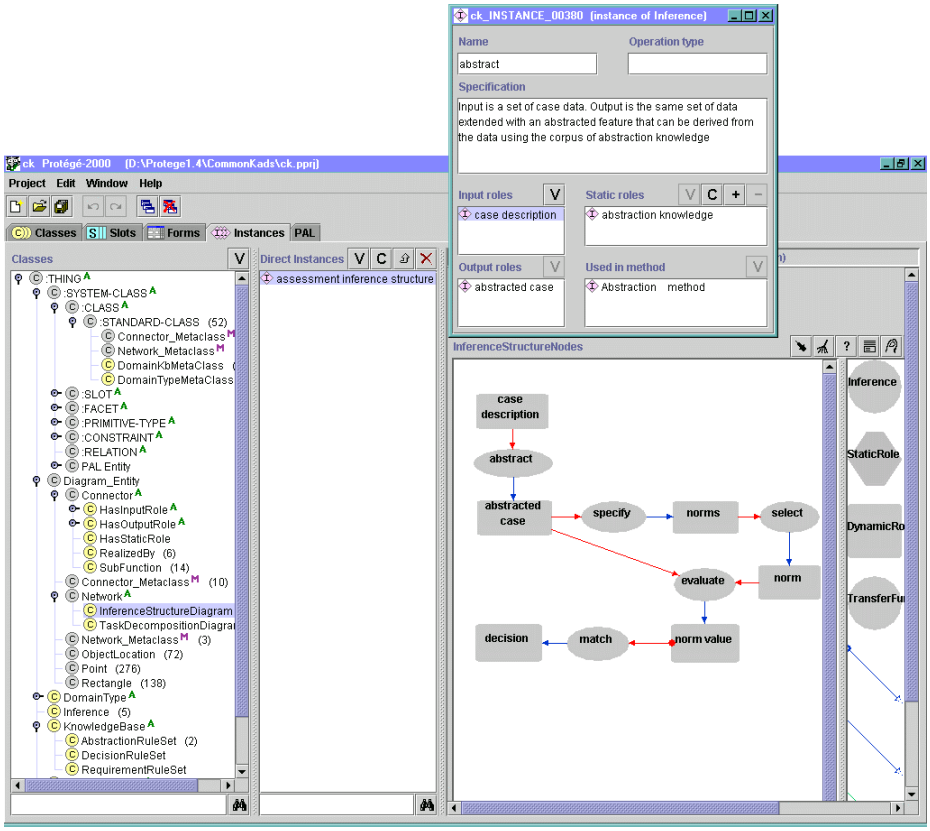

Fig. 2. Instance of an inference structure diagram for assessment. We defined the specific InferenceStructureDiagram as a subclass of the NetworkClass, thus also an instance of the NetworkMetaclass, that has additional slots to hold nodes and connectors. Our specific connectors are subclasses of ConnectorClass and instances of the ConnectorMetaclass. At the right, the form for acquiring an inference structure shows the instance created for an assessment task. It includes a palette of graphical elements from which the diagram can be constructed. The pop-up window shows the details (slot values) of the inference Abstract

node to a Inference or TransferFunction node. This way, when we create an instance of the InferenceStructureDiagram (see Fig. 2), we can add or create instances of the nodes, for example an inference abstract and a dynamic role case description, and link them with an instance of HasInputRole connector. Using the values of the slot pointers slots defined in HasInputRole, the diagram automatically fills-in the slot inputRoles of the Abstract instance with the instance value case description.

The use of the diagrams give rise to a large set of small problems and requirements with respect to their usage (symbol availability, user-interface behavior, etc.). We can see some small graphical differences (e.g., rounded rectangles, no border) when we

compare the inference structure in Fig. 2 with the original figure in the CommonKADS textbook [11, p. 136] (see Fig. 3). However, the basic mechanisms for form specification suited our purposes well.

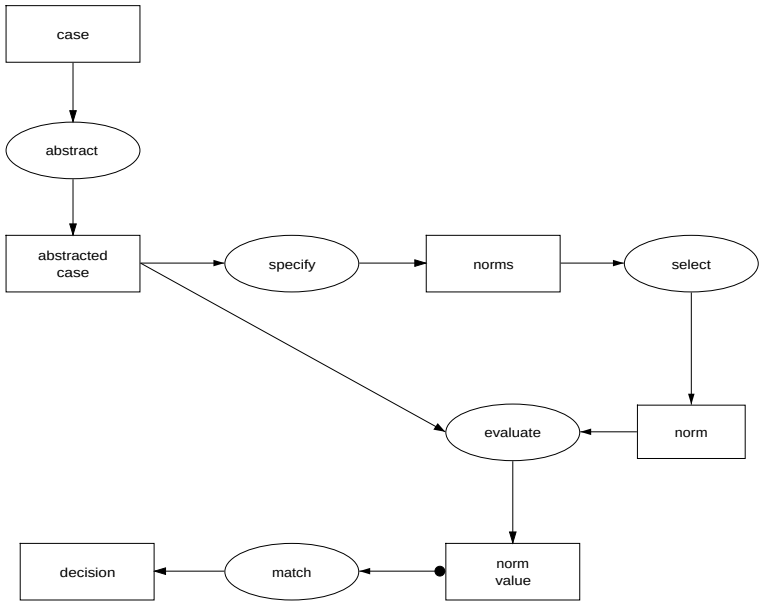

Fig. 3. Original figure of the assessment inference structure in the CommonKADS textbook

3.3 Domain schema

In principle, one can specify several different domain schemata in a knowledge model. For the moment we have limited the tool to a single schema. A schema contains a set of definitions of concepts, relations and rule types. A **concept** is represented as a class. The attributes of the concept (which in CommonKADS, as in UML, always point to atomic values, and not to other concepts) are modelled as slots.

To ensure that the attributes of a concept are atomic, we defined a constraint on the DomainTypeMetaClass metaclass, which is the template for all domain-modeling constructs such as the class Concept (see below). The constraint specifies that all subclasses of Concept (which define domain-specific types, such as Applicant) should have their template slots restricted to primitive (atomic) value type. The following formula is the constraint expressed using the language provided in Protégé:

```
(defrange ?dtype :FRAME DomainTypeMetaClass)
(defrange ?att   :FRAME :STANDARD-SLOT)
```

```
(forall ?dtype (forall ?att
   (=> (and (subclass-of ?dtype Concept)
            (template-slot-of ?att ?dtype))
       (allowed-slot-value-type ?dtype ?att :PRIMITIVE-TYPE))))
```

Fig. 4 shows an example concept Applicant, i.e a person applying for a house. The slots describe attributes of the applicant that can be used for assessment purposes.

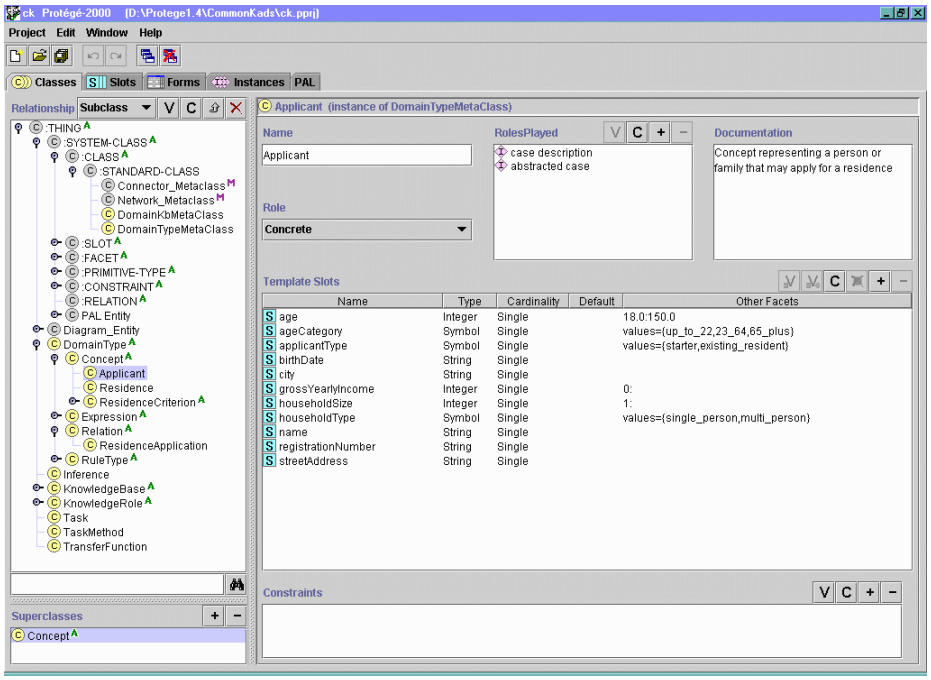

Fig. 4. Specification of the concept Applicant. The figure illustrates the three-level specification of the CommonKADS knowledge model. The domain-specific class Applicant is both a subclass of the domain-modeling construct Concept, and an instance of the special-purpose DomainTypeMetaClass metaclass (shown in the :CLASS subtree). Therefore it has an additional own slot rolesPlayed that can be filled-in in the right hand form. Individual applicants are instances of Applicant class, that have specific values for the template slots age, name, etc.

In Fig. 4 we can also see the class-instance issue discussed in Sec. 3.1. The class Applicant is at the same time a subclass of of Concept and an instance of the metaclass DomainTypeMetaClass. This metaclass has a slot rolesPlayed. The fillers of this slot are the knowledge roles that refer to this domain type (here: case description and abstracted case).

A relation in CommonKADS is a “first-class citizen”, meaning that it can have attributes. As explained in Sec. 2.1, slots in Protégé-2000 are also first-class objects, that are instances of a metaslot class. However, slot frames cannot form a slot hierarchy. Therefore, we do not model relations as slots but as domain classes (instances of DomainTypeMetaClass). Each “relation” class should have at least two slots that points to the object types being related (the relation “arguments”). These object types can be concepts and relations, meaning that higher-order relations are allowed (cf. the notion of “association class” in UML). The name of the slot pointing to the argument defines the role the argument plays in the relation. Other slots may be added to define attributes of the relation. For example, the relation between an Applicant and a Residence s/he applied for can be modelled as follows (simplified syntax, ’%’-sign is used as comment character):

```
CLASS ResidenceApplication SUBCLASS-OF Relation
  SLOT  applicant       -> Applicant   % relation argument
  SLOT  residence       -> Residence   % idem
  SLOT  applicationDate -> string      % relation attribute
  SLOT  eligible        -> boolean     % idem
```

The representation of a **rule type** is more complex. Rule types model relations between *expressions about* concept/relation slots. Therefore, we first have to define the notion of “expression”. We decided to model an expression as a class with four slots:

1. The concept/relation involved in the expression.
2. The possible slots involved. This should be an existing slot of the concepts/relations involved.
3. An operator such as `equal`, `not-equal`, `greater`. The set of legal operators depends on the slot involved in the expression.
4. The value: this should be a legal value for the slot involved.

With this expression construct we can define a class ApplicantExpression. An example instance of this expression could have the following slot values:

```
class   = Applicant
slot    = age
operand = greater
value   = 22
```

The intended interpretation of the expression instance is that the age of the applicant should be higher than 22.

Fig. 5 shows the definition of the Expression class. Besides the usual value-type restrictions on slots we specified constraints to express the above definition of an expression. We defined a first constraint to restrict the value of the slot involved in an expression (the value of the slot slot) to existing template slots of the class involved (the class value of the class slot). We defined a second constraint to ensure that the value involved in an expression (the value slot) is legal for the slot involved (the slot value of the slot slot at the class value of the class class).

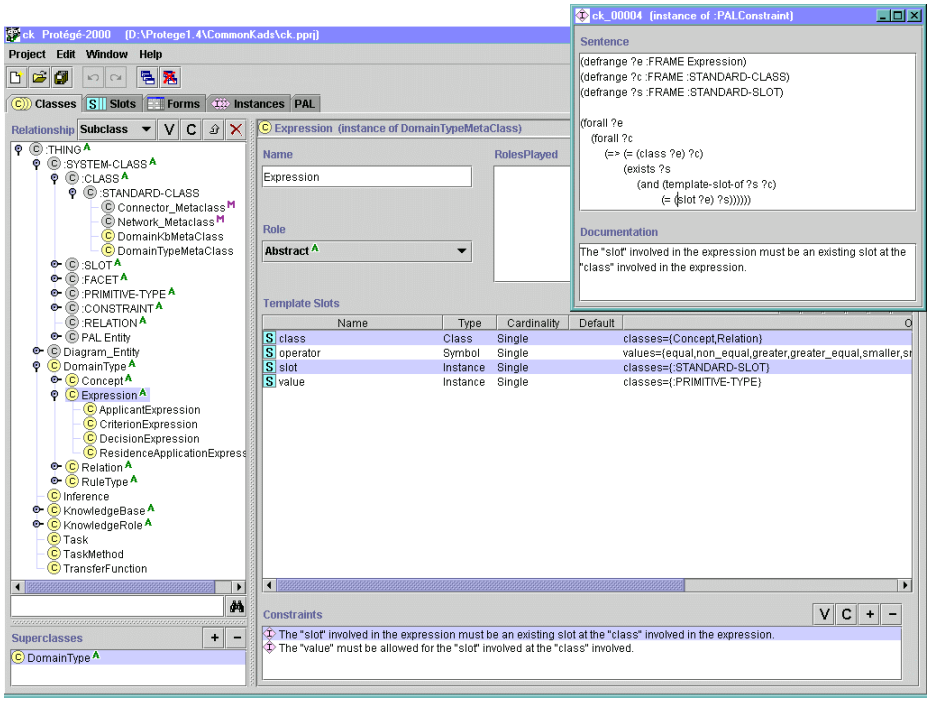

Fig. 5. Definition of the class Expression as an instance of DomainTypeMetaClass. The four slots that are attached to it have additional constraints on their value (bottom right). First, the slot slot involved in the expression should be an existing slot of the class involved. The KIF-based formula for this constraint is displayed in the pop-up window: it defines the range for the variables and the actual sentence of the constraint. A second constraint specifies that the value involved in the expression should be a valid value for the slot attached to the class involved

Constraints can subsequently be used to check the validity of expression instances. Fig. 6 shows how the constraint engine detected an instance of Expression that does not satisfy the first constraint.

We can now model a rule type as a relation between a set of expressions which form the antecedent of the rule and a set of expressions which constitute the consequent. A sample definition of the rule type ApplicantAbstraction is shown in Fig. 7.

The metamodel for the definition of rule types is summarized in Fig. 8. This figure uses a UML class-diagram notation. In the next subsection we see examples of the actual rule instances, and how we can define specialized forms for entering rules of a particular type.

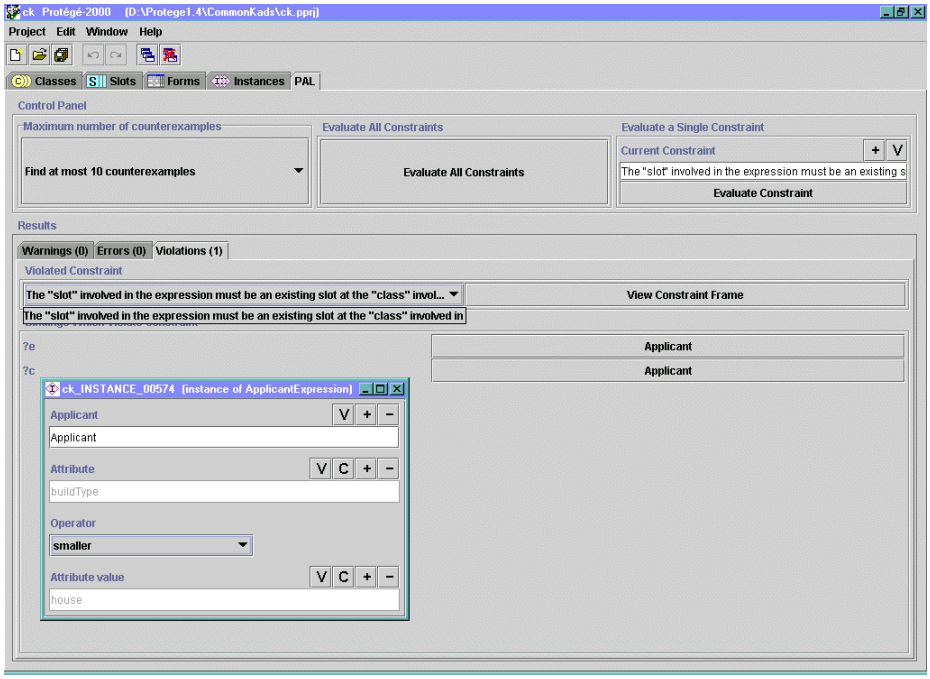

Fig. 6. Example of a Violated constraint in the knowledge base. A special tab in Protégé enables the user to trigger the constraint-checking engine on the knowledge base. Here, the constraint on the value of the slot involved in an expression has been checked for the knowledge base. An instance of Expression violates the constraint. Indeed, as the pop-up window shows, the value for slot is set to buildType, which is not a slot attached to Applicant (see Fig. 4). Note that the display names for slot, class and value have been customized to more meaningful names on the form for ApplicantExpression

3.4 Knowledge bases

In CommonKADS there is not one large knowledge base. Instead, several knowledge bases are defined. Each knowledge base contains instances of a designated set of domain types. Most commonly, knowledge bases contain instances of rule types, i.e. the actual "rules".

Knowledge bases are modelled as classes with a slot pointing to the rule types or other instances to be included in the knowledge base. Fig. 9 shows an instance of a knowledge base AbstractionRuleSet. This particular knowledge base contains two rules, both concerning a simple abstraction in the sample application.

Using Protégé's set of built-in user interface elements in a customized way, we defined a specialized form for acquiring instances of KnowledgeBase classes, shown in Fig. 9. First, we customized the form for acquiring RuleType instances to display its antecedent and consequent slots as rows that enable in-place editing of slot values. Then, we "included" this RuleType form in the form for KnowledgeBase instances. This

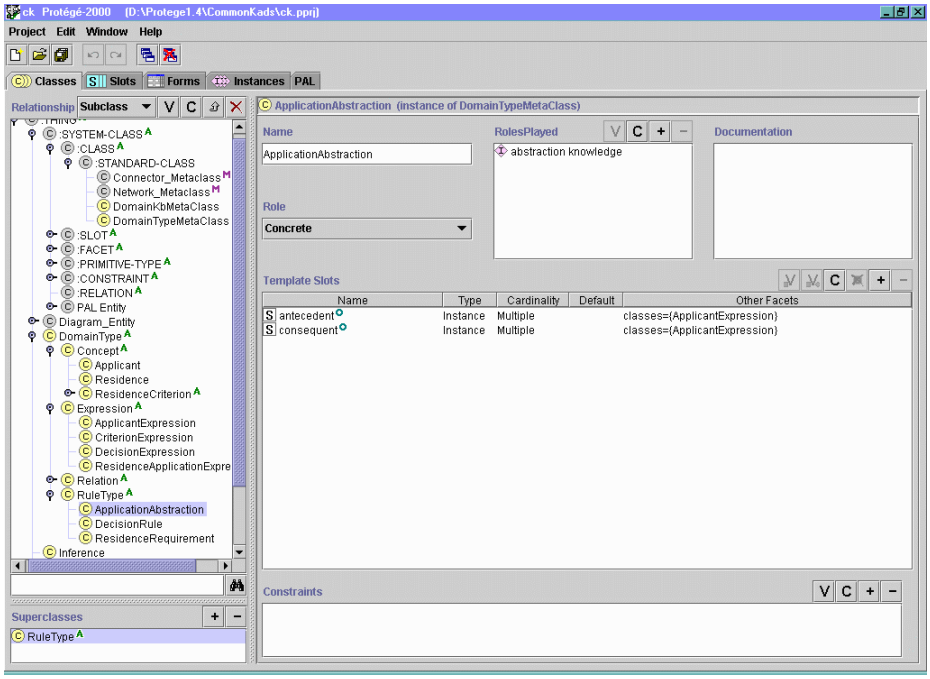

Fig. 7. Specification of the rule type ApplicantAbstraction. The inherited slots antecedent and consequent are specialized for ApplicantAbstraction so that their value type is restricted to instances of ApplicantExpression

enables the user to browse and create knowledge bases in a synthetic way, acquiring their rules and the expressions that form the rules immediately from the same form.

4 Discussion

This case study does not provide us with a formal evaluation. Still, a number of remarks can be made. It should be noted that the remarks are made from the perspective of using Protégé as a "metaCASE" tool. This typically requires stretching the possibilities of a tool to its limits. A standard case study using Protégé only directly as an ontology-editing tool would have given rise to different remarks.

In the discussion below on the strong and weak points of Protégé we refer back to the three user types mentioned in the introduction: tool builder, knowledge engineer, and domain specialist.

4.1 Strong points

The main strong point of Protégé is that it supports at the same time tool builders, knowledge engineers and domain specialists. This is the main difference with existing

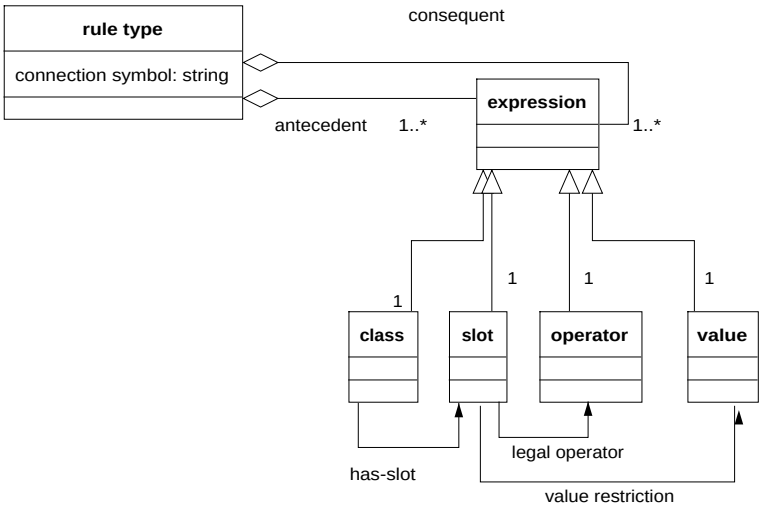

Fig. 8. Metamodel of a rule type (UML notation)

tools, which are typically targeted at the knowledge engineer and lack flexibility for metamodelling. This latter feature makes it easier to adapt Protégé to new requirements and/or changes in the model structure.

For the tool builder Protégé combines an expressive framework with a consistent way of generating editor interfaces (the forms). As mentioned, the metaclass mechanism in which classes can be modelled as "real" instances of metaclasses is an indispensable feature. Also, the fact that classes (as opposed to instances) can serve as the range of a slot is a useful feature (although this is also common in other languages). Another positive point is the time required to build a tool. Constructing a first version is a matter of a few hours. In this case study a number of revisions were made, but this is more or less intrinsic to the complexity of metamodelling in general.

Although still preliminary in terms of scope and user interface, an important feature for the tool builder is the possibility to define constraints on the classes and slots being defined. This provides a means for the knowledge engineer to check model completeness and correctness. In the context of this case study we specified a subset of the constraints for CommonKADS knowledge models[8], but it should be straightforward to generate a full set. In particular, we want to express the third constraint on expressions (see Sec. 3.3), to ensure that the set of legal operators offered on a certain type of expression (subclass of Expression) is suitable for the type of slot involved in the expression.

It is to note that during construction the models are *by definition* incomplete or even inconsistent. Therefore, a verification mechanism should preferably be explicitly invoked by a knowledge engineer, and not be done automatically by the tool. The fact

[8] Note for the reviewers: this is mainly due to the fact that constraints became available at the end of the case study. The final paper will reflect on the full set of constraints

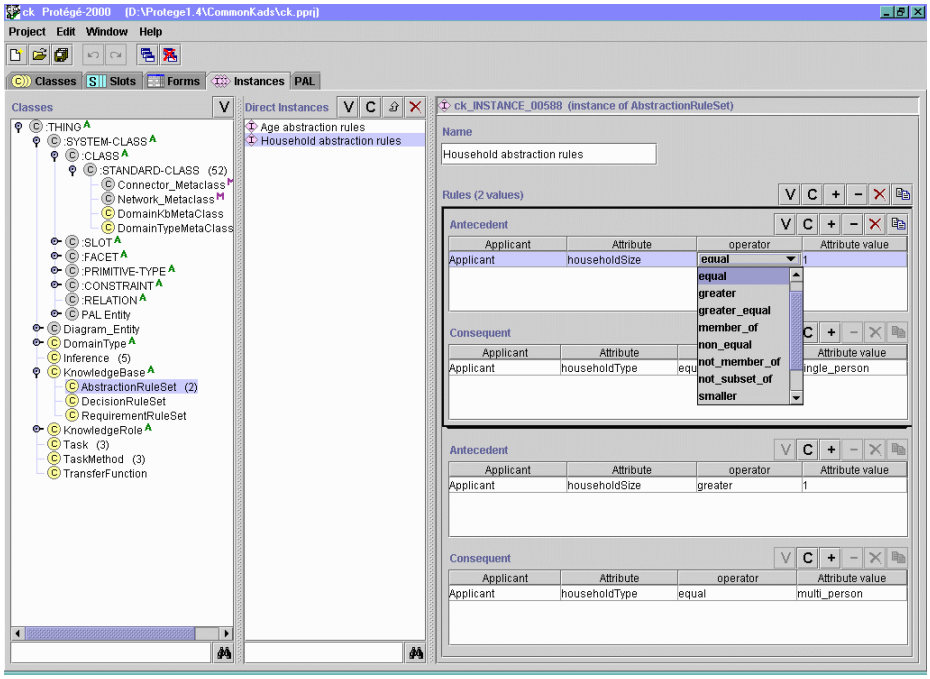

Fig. 9. Example knowledge base with two abstraction rules. The form for acquiring instances of KnowledgeBase has been highly customized to present a synthetic view of the rules contained in a knowledge base by way of container and table graphical elements. In-place editing enables to enter the values of the slots for a specific instance of Expression (that forms a RuleType) directly from this form

that the tool offers this approach of decoupling the acquisition phase from the constraint checking phase is a strong point. It should go further in enabling the check constraints step by step, for example locally to a class. The constraints we created in our modeling study range over class seen as instances of metaclasses (e.g., Expression, Concept) and restrict or bind the value-type of slots (rather than the value itself). That is again an important aspect of metamodelling.

A nice feature for the knowledge engineer is that it proved possible to generate diagrammatic editors close to the representation used in CommonKADS. The additional requirements here are really on a very detailed, uninteresting level, having to do with the availability of particular graphical symbols. labels, etc. Also, the RDF support is potentially a positive feature of Protégé, because it makes import into and export from the tool feasible in a standard format. This is traditionally problematic for CASE tools supporting only a proprietary format.

Finally, it should also be noted, that the fact that the tool never crashed during this case study also greatly helped in creating user confidence (especially with the first author).

4.2 Weak points and opportunities for improvement

Currently, Protégé only has a simple inclusion mechanism for importing definitions. In order to make use of libraries of existing partial models that the user can specialize and adapt, a more refined import/export mechanism for sets of class/form/instance definitions is required.

Another drawback is the fact that a single interface is used for all types of users. In particular for the domain specialist, it should be possible to create a stand-alone interface with just the forms for entering domain knowledge. This ensures that the domain specialist does not get confused by internal details. The same could hold, to a lesser extent, for the knowledge engineer.

At the moment, Protégé is not able to handle the automatic filling of inverse slot relationships, for example the domainMapping and rolesPlayed slots of respectively knowledge roles and domain types. This would be a useful extension, as it prevents redundancy and omissions in the interface for the knowledge engineer.

The expressiveness of the predefined forms could be extended in a number of ways. For example, the diagram widget could be extended to enable the creation of UML-type diagrams [2]. For CommonKADS this would open the possibility to create domain-schema diagrams, which use the UML conventions. It could also be a powerful way to set and visualize constraint links among classes, such as the way it is represented in Fig. 8." Another advantage would be that it makes Protégé potentially usable as a CASE tool for object-oriented analysis.

Rule types proved to be difficult to represent. The negative effect is mainly felt in the knowledge-elicitation interface created for the domain specialist. The somewhat contrived way of representing rule types diminishes the naturalness of the interface for entering rule instances in a knowledge base, as we saw in Sec. 3.4. It would be worthwhile to study more in detail the user-interface requirements for editing this type of expertise data, and adapt the tool accordingly. We could define a custom user interface component that would be more intuitive and synthetic to acquire rule bases, and would also check constraints on expressions locally.

4.3 Some final remarks

We plan to use the resulting CASE tool in an experiment in which a group of students in knowledge engineering at the University of Amsterdam uses the tool to construct knowledge models. The experiment is planned for the end of 2000. We will divide the students into two groups, one working with a baseline drawing tool, the other group with the knowledge-engineering interface of the Protégé-CommonKADS tool. This experiment will hopefully provide us with more precise data on usability.

Acknowledgments We thank the developers of the Protégé-2000 software tools, Ray Fergerson and William Grosso, for their valuable help and their promptitude to implement user requirements. Richard Benjamins contributed to the discussion on the setup of the tool. This work is partly supported by a grant from Spawar.

References

[1] A. Anjewierden, J. Wielemaker, and C. Toussaint. Shelley - computer aided knowledge engineering. *Knowledge Acquisition*, 4(1), 1992.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, Reading, MA, 1998.

[3] V.K. Chaudhri, A. Farquhar, R. Fikes, P.D. Karp, and J.P. Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings 15th National Conference on Artificial Intelligence (AAAI-98). Madison, Wisconsin*. AAAI Press/MIT Press, 1998.

[4] O. Corby and R. Dieng. The WebCokace knowledge server. *IEEE Internet Computing*, 3(6):38–43, November/December 199.

[5] N. Fridman Noy, R. W. Fergerson, and M. A. Musen. The knowledge model of Protégé-2000: combining interoperability and flexibility. SMI technical report, Stanford University, School of Medicine, 2000. Submitted for publication.

[6] M. R. Genesereth and R. E. Fikes. Knowledge interchange format version 3.0 reference manual. Report Logic 92-1, Logic Group, Stanford University, California, 1992.

[7] W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, and M. A. Musen. Knowledge modeling at the millennium: The design and evolution of Protégé-2000. In *12th Banff Workshop on Knowledge Acquisition, Modeling, and Management. Banff, Alberta*, 1999.

[8] O. Lassila and R. R. Swick. Resource description framework (RDF) model and specification. W3C recommendation, W3C Consortium, 22 February 1999. URL: http://www.w4.org/TR/1999/REC-rdf-syntax-19990222.

[9] Mark A. Musen, L. M. Fagan, D. M. Combs, and E. H. Shortliffe. Use of a domain-model to drive an interactive knowledge-editing tool. *Int. J. Man-Machine Studies*, 26:105–121, 1987.

[10] A. R. Puerta, J. Egar, S. Tu, and M. Musen. A multiple-method shell for the automatic generation of knowledge acquisition tools. *Knowledge Acquisition*, 4:171–196, 1992.

[11] A. Th. Schreiber, J. M. Akkermans, A. A. Anjewierden, R. de Hoog, N. R. Shadbolt, W. Van de Velde, and B. J. Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology*. MIT Press, Cambrdige, MA, 1999.

[12] A. Th. Schreiber, B. J. Wielinga, J. M. Akkermans, W. Van de Velde, and A. Anjewierden. CML: The CommonKADS conceptual modelling language. In L. Steels, A. Th. Schreiber, and W. Van de Velde, editors, *A Future for Knowledge Acquisition. Proceedings of the 8th European Knowledge Acquisition Workshop EKAW'94*, volume 867 of *Lecture Notes in Artificial Intelligence*, pages 1–25, Berlin/Heidelberg, September 1994. Springer-Verlag.

[13] A. Th. Schreiber, B. J. Wielinga, R. de Hoog, J. M. Akkermans, and W. Van de Velde. CommonKADS: A comprehensive methodology for KBS development. *IEEE Expert*, 9(6):28–37, December 1994.

[14] P. Speel, N. R. Shadbolt, W. de Vries, P. van Dam, and K. O'Hara. Knowledge mapping for industrial purposes. In *Proc Twelfth Workshop on Knowledge Acquisition, Modelling Management (KAW'99)*, 1999.

[15] S. W. Tu, H. Eriksson, J. H. Gennari, Y. Shahar, and M. A. Musen. Ontology-based configuration of problem-solving methods and generation of knowledge acquisition tools: The application of PROTÉGÉ-II to protocol-based decision support. *Artificial Intelligence in Medicine*, 7(5), 1995.

[16] G. van Heijst, P. Terpstra, B. J. Wielinga, and N. Shadbolt. Using generalised directive models in knowledge acquisition. In Th. Wetter, K. D. Althoff, J. Boose, B. Gaines, M. Linster, and F. Schmalhofer, editors, *Current Developments in Knowledge Acquisition: EKAW-92*, Berlin, Germany, 1992. Springer-Verlag.