A Tool for Sharing Knowledge with Ontologies

Adrian Giurca

University of Craiova,

Faculty of Mathematics and Computer Science,

13 A.I.Cuza street, 1100, Craiova

giurca@inf.ucv.ro

ROMANIA

Abstract: The goal of this paper is to present a new tool called HKS that is designed to help users to collect information about various concepts and their attributes from HTML pages. This version of the tool should facilitate the manual extraction of information from the visualized pages via a graphical user interface and the use of techniques like drag & drop and copy & paste. The extracted information is used to populate the Knowledge Base/Data Base. The acquired knowledge is organized as a multiple inheritance graph of concept descriptions. The tool is also being capable to import/export concept descriptions from existing toplevel ontologies and create/update ontologies.

Keywords: AMS Classification (2000). Primary: 68T30, Secondary: 68P20

1. Introduction

The Semantic Web is the name given to the Internet of the future where inference services are supported. Tim BernersLee presented the following diagram (Figure 1) during a talk at the XML World 2000 Conference in Boston, Massachusetts to describe the infrastructure of the web which will support this vision. This diagram has become known as the Semantic Web Layer Cake.

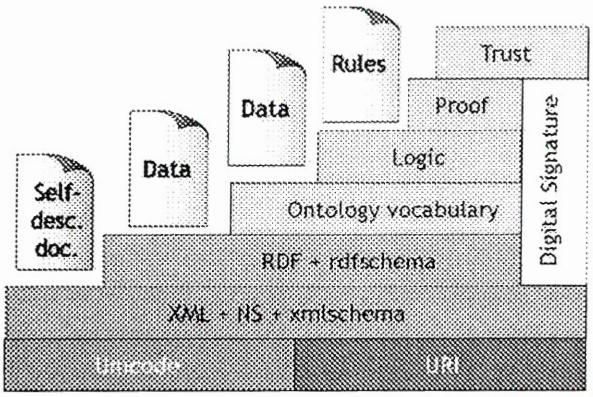


Figure 1: The Semantic Web Layer Cake (BernersLee,2000)

In the Semantic Web vision, unambiguous sense in a dialog among remote applications or agents can be achieved through shared reference to the ontologies available on the network, albeit an always changing combination of upper level and domain ontologies. We just have to assume that each ontology is consensual and congruent with the other shared ontologies (e.g., ontologies routinely include one another). The result is a common domain of discourse that can be interpreted further by rules of inference and application logic.

The creation of joint ontologies for a number of agents is a challenging task. Distributed development of ontologies e.g. needs tools for synchronizing between a number of agents. But to acquire an ontology from only one agent is also difficult, because is means to make explicit something that is usually just implicit.

An ontology is typically built in more-or-less the following manner:

1. **Acquire domain knowledge**

Assemble appropriate information resources and expertise that will define, with consensus and consistency, the terms used formally to describe things in the domain of interest. These definitions must be collected so that they can be expressed in a common language selected for the ontology.

2. **Organize the ontology**

Design the overall conceptual structure of the domain. This will likely involve identifying the domain's principal concrete concepts and their properties, identifying the relationships among the concepts, creating abstract concepts as organizing features, referencing or including supporting ontologies, distinguishing which concepts have instances, and applying other guidelines of your chosen methodology.

3. **Flesh out the ontology**

Add concepts, relations, and individuals to the level of detail necessary to satisfy the purposes of the ontology.

4. **Check your work**

Reconcile syntactic, logical, and semantic inconsistencies among the ontology elements. Consistency checking may also involve automatic classification that defines new concepts based on individual properties and class relationships.

5. **Commit the ontology**

Incumbent on any ontology development effort is a final verification of the ontology by domain experts and the subsequent commitment of the ontology by publishing it within its intended deployment environment.

Our main goal is to provide a tool, which may furnish the main functionality for build ontologies and populate them with knowledge acquired from the web.

After 2000 there are many tools involved in creation and editing ontologies. Here we present four tools which are, in our opinion, the most important:

- **Protege2000** [10, 24] - an ontology editor and a knowledge-base editor. Protégé is also an open-source, Java tool that provides an extensible architecture for the creation of customized knowledge-based applications. Protégé's OWL Plug-in [23] provides support for editing Semantic Web ontologies.

- **OilEd** [3,4,20] - is an ontology editor allowing the user to build ontologies using DAML+OIL [8,14]. It does not provide a full ontology development environment - it will not actively support the development of large-scale ontologies, the migration and integration of ontologies, versioning, argumentation and many other activities that are involved in ontology construction. Rather, it is the "NotePad" of ontology editors, offering enough functionality to allow users to build ontologies and to demonstrate how we can use the FaCT reasoner to check those ontologies for consistency.

- **OntoEdit** [21] – is an ontology-engineering environment that uses a powerful ontology model to store the conceptual model of ontology.

- **WebOnto** [26] - was designed to support the collaborative browsing, creation and editing of ontologies without suffering from the interface problems described in the previous section. In particular, WebOnto was designed to provide a direct manipulation interface displaying ontological expressions using a rich medium. WebOnto was aimed to be easy-to-use, yet have facilities for scaling up to large ontologies.

For a large survey about ontologies tools the reader can consult the OntoWeb report [22]. In the Section 4 of this paper we present a short comparison between these tools and the HKS tool.

2. Using F-Logic in HKS

In the papers [5] and [6] we proposed a mathematical model for representing knowledge in ontologies. The base language used is F-Logic, [9]. F-Logic is a deductive, object oriented database language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modeling capabilities supported by the object oriented data model. Also I use the experience of the Florid [15], project at the university of Freiburg. The mathematical definitions of the HKS Model are shortly described in the Appendix.

This section briefly describes how we use F-Logic in the HKS1.0 system.

2.1 Objects, their Properties and Structure

Objects model real world entities and are internally represented by object identifiers which are independent of their properties. According to the principles of object oriented systems these object identifiers are invisible to the user. To access an object directly the user has to know its object name. Every object name refers to exactly one object. Following the object oriented paradigm, objects may be organized in classes. Furthermore, methods represent relationships between objects. Such information about objects is expressed by F-atoms. Instead of giving several individual atoms, information about an object can be collected in F--molecules. For example, the following F-molecule denotes that `printer1` is a printer whose product price is `printprice1` and his name is `HP Printer`.

```
printer1:Printer[ProductPrice=>>printprice1;
  PrinterName=>>"HP Printer"].
```

This F-molecule may be split into several F-atoms:

```
printer1:Printer.
printer1[ProductPrice =>> printprice1] AND
printer1[PrinterName=>>"HP Printer"].
```

For F-molecules containing a multivalued method, the set of result objects can be divided into singleton sets (recall that our semantics is multivalued, not setvalued). For singleton sets, it is allowed to omit the curly braces enclosing the result set.

```
procesor1:Procesor. procesor1[isModel=>>{"AMD", "INTEL"}].
```

has the same semantics with

```
procesor1:Procesor.
procesor1[isModel=>>"AMD"]. procesor1[isModel=>>"INTEL"].
```

2.2. Classes and Methods

Isa-F-atoms state that an object belongs to a class, subclass-F-atoms express the subclass relationship between two classes. A single colon and a double colon denote class membership and the subclass relation, respectively. The following example defines a `Printer` class and two instances `printer1` and `printer2`. The `Printer` class is a subclass of `Product` class. The `Printer` class has the multivalued method `hasDummyPrinter` which return a set of objects from the `Printer` class.

```
Printer::Product.

Printer[

hasDummyPrinter=>>Printer;

PrintingResolution=>>PrinterResolution;

PrinterName=>>STRING ]. printer1:Printer. printer2:Printer.
```

2.3. Predicate Symbols

In our model, predicate symbols are used in the same way as in predicate logic, e.g., in Datalog, thus preserving upward compatibility from Datalog to F-Logic. A predicate symbol followed by one or more id-terms separated by commas and included in parentheses is called a P-atom to distinguish it from F-atoms. Next example shows some predicates. The last predicate consists solely of a 0ary predicate symbol. Those are always used without parentheses.

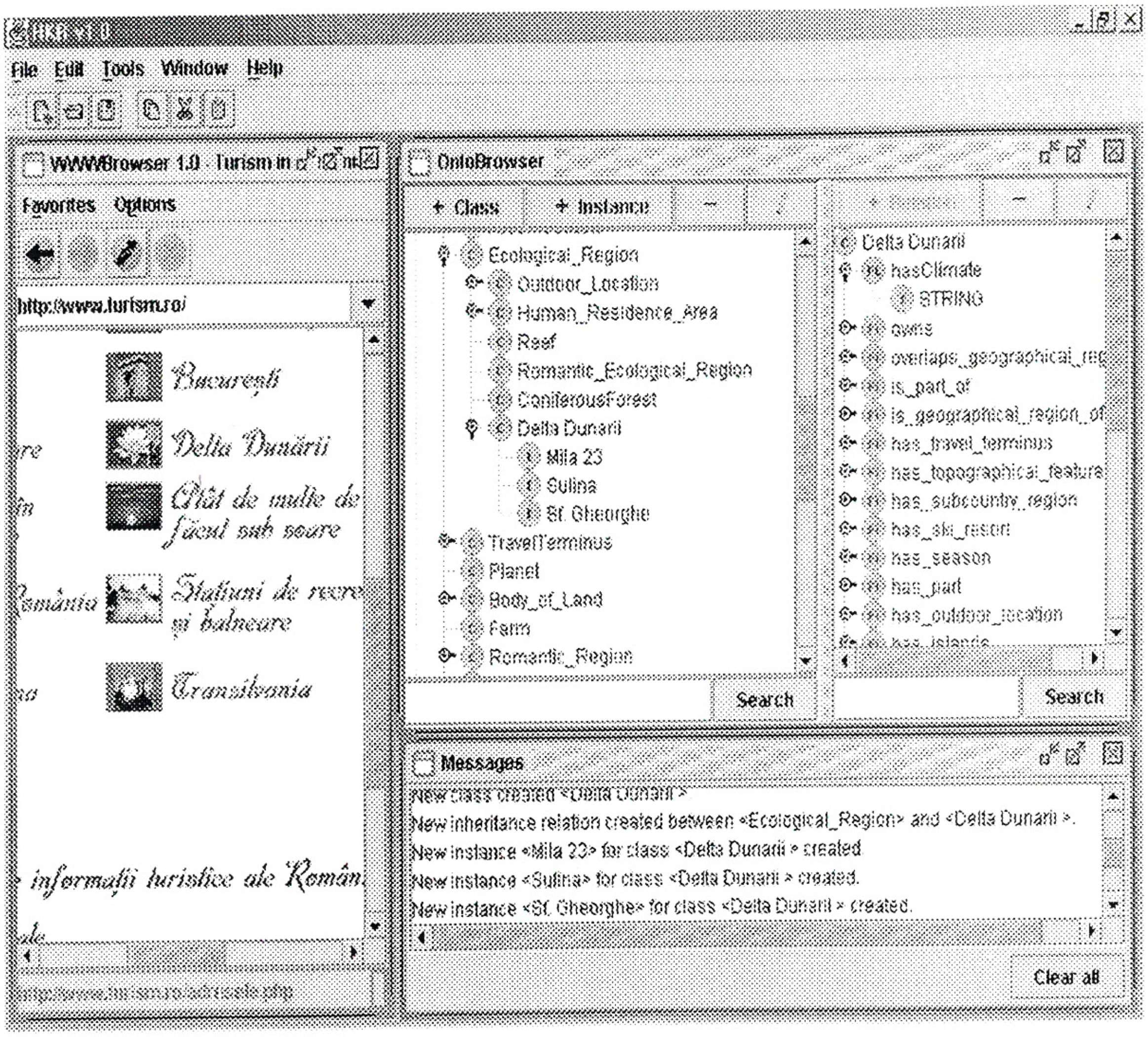
```
office_system(computer1, printer1).

peripheral(printer1).

true.
```

3. Architecture Overview

This section is devoted to describe the architecture of the HKS tool. Mainly, we describe the core packages of the HKS and the analysis level of UML class-diagrams. The Figure 5 is a snapshot of the HKS1.0 tool.



New class created <Delta Dunarii >
New inheritance relation created between <Ecological_Region> and <Delta Dunarii >.
New instance <Mila 23> for class <Delta Dunarii > created
New instance <Sulina> for class <Delta Dunarii > created.
New instance <St. Gheorghe> for class <Delta Dunarii > created.

Figure 2: HKS1.0 Ontology tool snapshot

3.1. Packages

This section present the main architecture of the tool. This tool is developed on a four core packages:

- The ONTOMODEL package business knowledge representation
- The MODEL package low level knowledge representation
- The BROWSER package a HTTP browser with drag and drop capabilities.
- The GUI package user interface, with drag and drop capabilities.

We use also some additional libraries packages:

- JAXP [16] - The Java API for XML Processing (JAXP) that applications to parse and transform XML documents independent of a particular XML processing implementation.

- CUP [13]- LALR parser generator for java.
- LOG4J [18]- Apache Logging Services Project an extensible logging library project for Java.

3.2. Knowledge Representation Layer. The Model/OntoModel core packages

The aim of this section is to describe the main concepts that are used to completely describing compositional structures like `Product`, `Printer` etc.

3.2.1 Definitions

We use a knowledge representation based on a directed graph structure.

Definition 1. Let *Nodes* a nonempty set of elements called *nodes* (like in a graph representation). The structure of a node is:

(id, name)

where *id* is an unique ID for each node and *name* is the name of the node (its label). There are two kinds of nodes:

(1) *OntoNode* - a node used to represent a class description of our ontology. Let *OntoNodes* be the set of all ontonodes.

(2) *OntoInstance* - a node used to represent an instance of a class in our ontology. Every *OntoInstance* node has a correspondent *OntoNode*. Let *OntoInstances* be the set of all ontoinstances.

For convenience we assume that *Nodes* = *OntoNodes* $\cup$ *OntoInstances* and *OntoNodes*, *OntoInstances* are disjoints sets.

Definition 2. Let *Relations* be a nonempty set of elements (directed edges in a graph representation) called *relations*. A relation has the structure: *(id, name, lef tArg, rightArg),* where *id* is the unique ID for each relation, *name* is the name of the relation, *leftArg* and *rightArg* are nodes, respectively the left and the right argument of the relation. There are three kinds of relations:

(1) *ROntoNode* relation a relation where *leftArg*, *rightArg* $\in$ *OntoNodes*.

(2) *RontoNodeInstance* relation a relation where *leftArg* $\in$ *OntoNode*s and *rightArg* $\in$ *OntoInstances*.

(3) *ROntoInstance* relation a relation where *leftArg, rightArg*$\in$ *OntoInstances*.

Clearly, all kinds of relations are two pair disjoints.

In the Figures 3 below we present the MODEL package which represent nodes and relations.

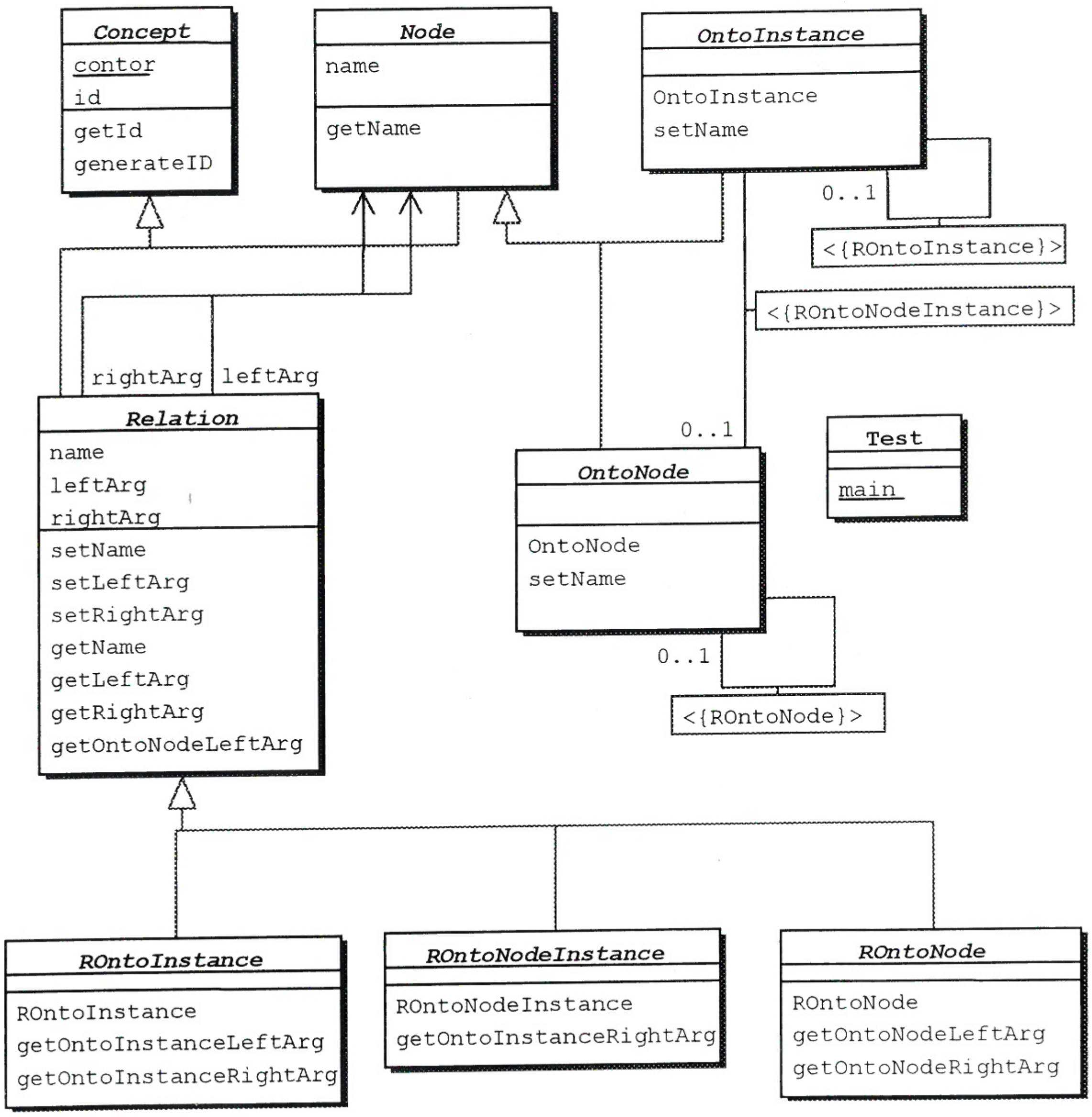


Figure 3: UML Class Diagram: Model Package

The Figure 4 presents a summary of the ONTOMODEL package.

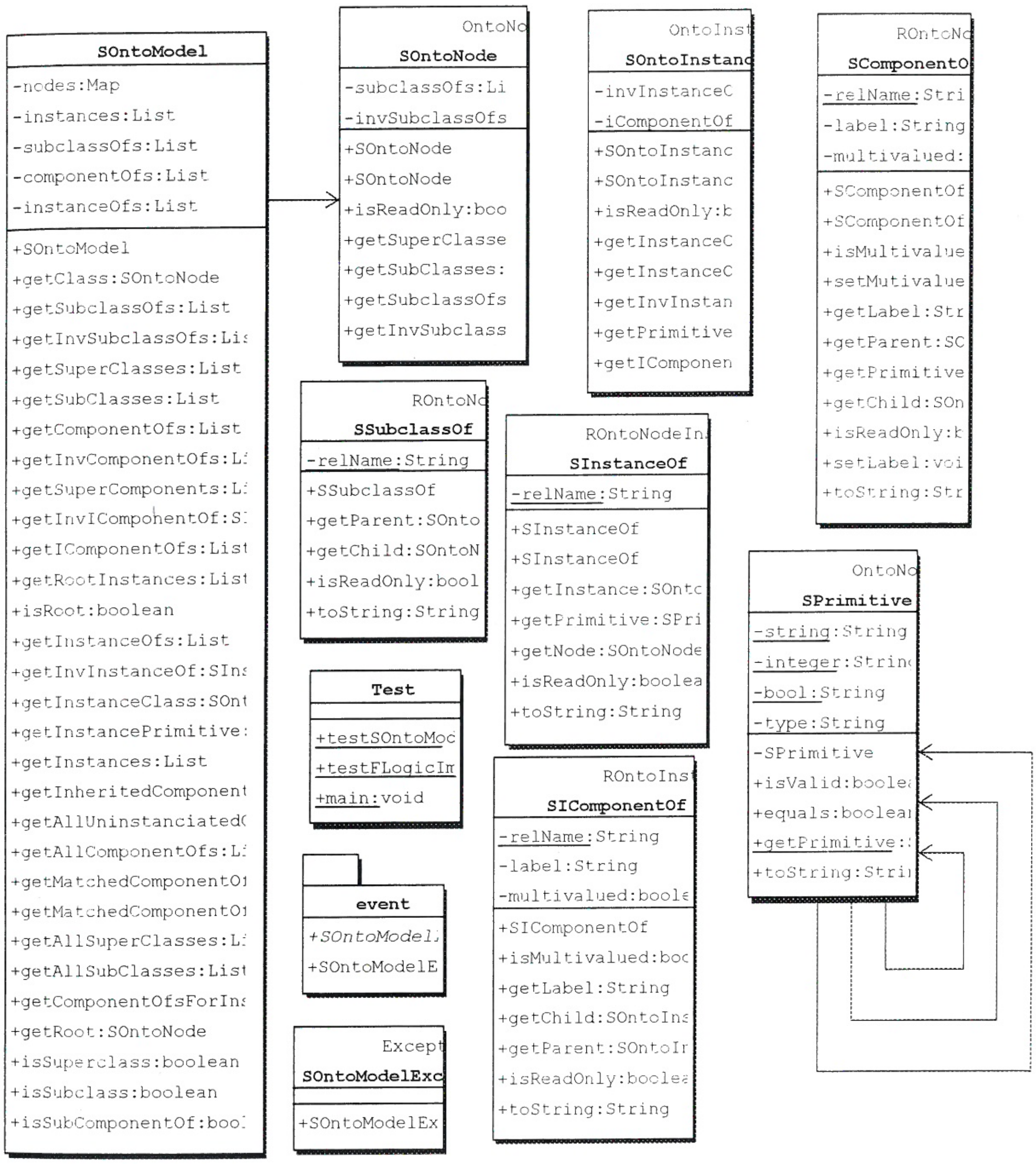


Figure 4: UML Class Diagram: OntoModel

4 HKS functionality

In a nutshell, you can use HKS for the following:

- Class modeling. HKS provides a graphical user interface (GUI) that models classes (domain concepts) and their attributes and relationships.
- Instance editing. From these classes, HKS enable you or domain experts to enter valid instances.

- Model processing. HKS has an internal model that helps you to capture semantics and obtain a logical behavior.
- Model exchange. The resulting models (classes and instances) can be loaded from F-Logic and saved in F-Logic and RDF.

The most important panel when you start a project is OntoBrowser, shown in Figure 2. In HKS and many other knowledge-modeling tools, classes are named concepts from the domain that can have attributes and relations. HKS classes are comparable to Java or UML classes, but without attached methods. Classes can be arranged in an inheritance hierarchy, which displays in the tree panel in the left part of the OntoBrowser panel. The properties of the tree's selected class display in the right part of the OntoBrowser panel. HKS supports multiple inheritance, and classes are imported (cannot be modified) or manually created (can be modified). Unlike in Java, both classes can have instances.

In HKS, classes attributes and their relations are all relations. A relation has a name and a value type. HKS supports the primitive value types BOOLEAN, INTEGER, FLOAT, and STRING, which are handled, like they are in Java. For example, you can define the class *Printer* and assign a relation called *name* to it with STRING as the value type. Additionally, a relation has cardinality – now just single-valued relation or multi-valued relation. Apart from primitive values, relations can also refer to the model's classes. A simple search facility is provided (at the bottom of the OntoBrowser panel) both for classes, instances and relations.

So the HKS relations are very similar to conventional object-oriented attributes and relations. However, some important details make HKS relation definitions richer than most object-oriented concepts. A main difference is that a HKS relation can attach to multiple classes.

Another major difference is that you can specify constraints on HKS relation values. Constraints restrict a relation's range of allowed values. One of these constraints restricts a relation's cardinality. You can specify single (i.e. a single value is accepted) or multiple (a list of values is accepted). This feature is similar in UML, where you can define cardinalities like [0..1] or [0..*]. HKS also allows you to define inverse of a relation. Also it provide an internal reasoning module using the reflexivity and role-hierarchy axioms. All these constraints help you build correct domain models, because HKS can display an instance's invalid values.

In HKS the user can acquire values, using drag and drop facilities, directly from the web via the HKS HTTP Browser. Of course, it is possible to acquire values manually.

A storage plug-in is a nonvisual module that saves and loads models in a certain file format. HKS currently supports F-logic and RDF file storage formats. As a plus, in HKS it is possible to include an ontology fragment written in F-Logic in the current ontology. Below we present a summary of the functions of the HKS tool.

Function 1: Creating/Open a new Ontology

This function is part of scenario KA1 in [1] and part of scene 1 in [7]. The knowledge engineer (KE) will use this function to create a new ontology i.e. a Hierarchical Knowledge Base (HKB) for a new e-commerce domain. The new ontology will be created either from scratch (not very often) or using a library of existing, toplevel ontologies (most often). This function will allow the KE to select the toplevel ontology on which the new HKB is based.

Function 2: Open an Ontology

The KE might decide to temporary save his work on the current HKB and continue it at a later time. So he is faced with the problem of loading a previously saved HKB. This function will allow the KE to select a previously saved HKB and load it into HKS. This function is not mentioned in documents [1] and [7].

Function 3: Managing Classes

This group of subfunctions will allow the KE to select/create/edit/delete classes.

Subfunction 3.1.: Select a class

The KE is faced with the problem of selecting a class from the HKB to perform some actions on it (i.e. edit, add instance, delete, add a new class as subclass...). This subfunction will allow the KE to perform this selection. In the selection process he must distinguish between own classes and imported classes. The imported classes cannot be modified.

Subfunction 3.2.: Create a new class

The KE can manually create new classes. A class has exactly one parent class (either an imported class or an own class). The KE will first select the parent class and then create the new class. The KE must provide a unique name for the new class. The HKS tool will provide a default name. This subfunction is necessary as in scenario KA2 from [1].

Subfunction 3.3: Edit a class

The KE can manually add, update or delete components to/from a selected own class. The KE will first select the class and then perform the desired actions. Also, when a new class is created the user can perform any of the above actions.

Subfunction 3.4: Delete an own class

The KE can delete an own class (again, not imported) from the HKB. The deletion will cause the removal from the HKB of the subtree rooted at this class. This subtree includes also the instances of the classes, which are member of the subtree.

Function 4: Manage Instances

This group of subfunctions will allow the KE to select/create instances and fillin them by manual acquisition from the Web, also, edit/delete the instances.

Subfunction 4.1.: Select an instance

The KE is faced with the problem of selecting an instance from the HKB to perform some actions on it (i.e. edit or delete). This subfunction will allow the KE to perform this selection. In the selection process he must distinguish between own instances and imported instances. Note that this subfunction was not addressed in the documents [1] and [7].

Subfunction 4.2.: Create a new instance

The KE wants to create a new instance of a given class (either an imported or an own class). The KE will first select the class and then create a new instance for it. The KE can then fill in the relations of the new instance with values acquired from the Web. The acquired values must satisfy the type constraints of the corresponding relation.

Subfunction 4.3.: Manually edit an instance

The KE can manually update the relation values of an instance or create by manually editing new relation values for an instance. Note that the KE is not allowed to update or delete imported instances.

Subfunction 4.4: Delete an instance

The KE delete a non-imported instance from the HKB. The KE will select the instance using subfunction 4.1 and then delete it.

Function 5: Save the HKB

The KE may decide to save his current work on a HKB such that it would be possible to continue it later. For this purpose he will save the current HKB.

Function 6 Include/Import/Export an ontology

At definite moments in time the KE may include an existing ontology fragment (which will be imported) or export the HKB. Thus, when next time the KE will create a new HKB based on importing, the old HKB classes and instances will be considered in the new HKB as imported. This function is needed in scenario KA10 from [1].

Subfunction 6.1 Include/Import an Ontology

At definite moments in time the KE may include an existing ontology fragment in the existent open ontology. All included classes and instances will be treated as imported i.e. no update or delete operation is not allowed. This tool version can include ontologies which was written in F-Logic and RDF. The import subfunction of the tool is definite necessary because in the most of cases the KE use a toplevel ontology for creating his HKB. When we import an ontology the already open ontology must be closed (and eventually saved).

Subfunction 6.2 Export an Ontology

The export subfunction of the tool assumes an open ontology and provides a file in F-Logic or RDF format.

4. A Short Comparison and Future Work

This section is devoted to a short comparison between the HKS tool and the tools already presented in the first section of the paper.

We propose the following criteria of comparison:

CRITERIA 1: Modeling Features/Limitations - The representational and logical qualities that can be expressed in the built ontology.

	Built in inference with three F-Logic axioms on relations: reflexivity, inverse and role-hierarchy. Equivalent instances.
OilEd 4/12/02	DAML constraint axioms; same-class-as; limited XML Schema datatypes; creation metadata; allows arbitrary expressions as fillers and in constraint axioms; explicit use of quantifiers; one-of lists of individuals; no hierarchical property view.
OntoEdit 8/6/02	F-Logic axioms on classes and relations; algebraic properties of relations; creation of metadata; limited DAML property constraints and datatypes; no class combinations, equivalent instances.
Protégé-2000 4/10/02, 10/22/02	Multiple inheritance concept and relation hierarchies (but single class for instance); meta-classes; instances specification support; constraint axioms ala Prolog, F-Logic, OIL and general axiom language (PAL) via plug-ins.
WebOnto 5/1/02	Multiple inheritance and exact coverings; meta-classes; class level support for prolog-like inference.

CRITERIA 2: Base Language – The native or primary language used to encode the ontology.

	F-Logic
OilEd 4/12/02	DAML+OIL
OntoEdit 8/6/02	F-Logic
Protégé-2000 4/10/02, 10/22/02	OKBC model
WebOnto 5/1/02	OCML

CRITERIA 3: Web Support & {Use} - Support for Web-compliant ontologies (e.g., URIs), and {use of the software over the Web (e.g., browser client)}.

	No. It has a built in browser (with limited capabilities) for knowledge aquisition
OilEd 4/12/02	RDF URIs; limited namespaces; very limited XML Schema
OntoEdit 8/6/02	Resource URIs
Protégé-2000 4/10/02, 10/22/02	Limited namespaces; {can run as applet; access through servlets}
WebOnto 5/1/02	{Web service deployment site}

CRITERIA 4: Import Export Formats - Other languages the built ontology can be serialized in.

	This version only cans import/include from F-Logic and export to RDF, F-Logic.
OilEd 4/12/02	RDFS; SHIQ
OntoEdit 8/6/02	RDFS; F-Logic; DAML+OIL (limited); RDB
Protégé-2000 4/10/02, 10/22/02	RDF(S); XML Schema; RDB schema via Data Genie plug-in; (DAML+OIL backend due 4Q'02 from SRI)
WebOnto 5/1/02	Import: RDF; Export: RDFS, Ontolingua, OIL

CRITERIA 5: Graph View - The extent to which the built ontology can be created, debugged, edited and/or compared directly in graphic form.

	Graphical browsing of classes instances and relations. No graph view.
OilEd 4/12/02	Browsing Graphviz files of class subsumption only.
OntoEdit 8/6/02	Yes, via plug-in
Protégé-2000 4/10/02, 10/22/02	Browsing classes & global properties via GraphViz plug-in; nested graph views with editing via Jambalaya plug-in.
WebOnto 5/1/02	Native graph view of class relationships.

CRITERIA 6: Consistency Checks - The degree to which the syntactic, referential and/or logical correctness of the ontology can be verified automatically.

	Full consistency check. Every import and/or include resolve the consistency in the internal model of the ontology.
OilEd 4/12/02	Subsumption and satisfiability (FaCT)
OntoEdit 8/6/02	Yes, via OntoBroker
Protégé-2000 4/10/02, 10/22/02	Plug-ins for adding & checking constraint axioms: PAL; FaCT.
WebOnto 5/1/02	For OCML code

CRITERIA 7: Merging - Support for easily comparing and merging independent built ontologies.

	Accept including of the fragments ontology from F-Logic
OilEd 4/12/02	No
OntoEdit 8/6/02	Yes
Protégé-2000 4/10/02, 10/22/02	Semi-automated via Anchor-PROMPT.
WebOnto 5/1/02	No

CRITERIA 8: Information Extraction - Capabilities for ontology-directed capture of target information from content and possibly subsequent elaboration of the ontology.

	No. The knowledge acquisition is made via browser with drag and drop capabilities.
OilEd 4/12/02	No
OntoEdit 8/6/02	No
Protégé-2000 4/10/02, 10/22/02	No
WebOnto 5/1/02	(Available from OCML based tool MnM.)

The future work will be devoted to two main objectives:

- To define a general reasoning mechanism that can be implemented in inference engine. There is some research in this direction concerning the type of logic to modeling the semantic but we still have some uncertainties about a good inference engine.
- To obtain representations in various knowledge formats such as OIL, [3, 4, 8] and DAML+OIL, [14].

5. Appendix – The Mathematical Model of HKS

The HKS Knowledge Model, [5, 6] assumes a universe of discourse consisting of all entities about which knowledge is to be expressed. Each knowledge base may have a different universe of discourse. However, HKS assumes that the universe of discourse always includes the following *Data Types*: BOOLEAN, STRING, INTEGER, FLOAT and all *Data Type Objects* (i.e. constants) of these data types. Let T be the set of data types. To provide a precise and succinct description of the HKS Knowledge Model, we use the

Knowledge Interchange Format (KIF) [17] as a formal specification language. KIF is a first-order predicate logic language with set theory, and has linear prefix syntax.

Let C be the set of classes.

Definition 1 (Class and Class-Object). *Classes* are sets of entities. Each of the entities in a class is said to be a *class-object* of the class. An entity can be a class-object of multiple classes, which are called its types or parents. A class cannot be a class-object of a class.

Thus, the domain of discourse consists of class-objects and classes. The unary relation class is true if and only if its argument is a class and the unary relation class-object is true if and only if its argument is an individual. The following axiom holds:

($\Leftrightarrow$ (class ?X) (not (class−object ?X)))

Definition 2 (instance-of, type-of). The class membership relation called *instance-of* that holds between an class-object and a class is a binary relation that maps entities to classes. A class is considered to be a unary relation that is true for each instance of the class. That is:

($\Leftrightarrow$ (holds ?C ?I) (instance−of ?I ?C))

The relation *type-of* is defined as the inverse of relation *instance-of*. That is,

($\Leftrightarrow$ (type−of ?C ?I) (instance−of ?I ?C))

Definition 3 (subclass-of, superclass-of). The *subclass-of* relation for classes is defined in terms of the relation *instance-of*, as follows. A class C_s is a subclass of class C_S if and only if all instances of C_s are also instances of C_S. That is,

($\Leftrightarrow$ (subclass−of ?C_s ?C_S)(forall ?I ($\Rightarrow$(instance−of ?I ?C_s) (instance−of ?I ?C_S))))

The relation *superclass-of* is defined as the inverse of the relation *subclass-of*. That is,

($\Leftrightarrow$ (superclass−of ?C_S ?C_s) (subclass−of ?C_s ?C_S))

The *subclass-of* relation is transitive (i.e., If A is a subclass of B and B is a subclass of C, then A is a subclass of C.).

Properties are binary relations involving classes and/or data types.

Definition 4 (HKS Relation). A *HKS relation* is a binary relation between two classes or between a class and a data type. So, if p denote a HKS relation, then $p \subseteq C\times C$ or $p \subseteq C\times T$. A HKS relation has associated a set of meta-relations: *Cardinality*, *Inverse*, *Role Hierarchy*.

The *cardinality* meta-relation specifies the number of values that may be asserted for a component on a class. Our framework supports two cardinalities: *single cardinality* and *multiple cardinality*.

- If a HKS relation p has single cardinality, then $\forall$A,B, (A,B) $\in$ p we can assert exact one class-object of B.
- If a HKS relation p has single cardinality, then $\forall$A,B, (A,B) $\in$ p we can assert one or more class-objects of B.

The *inverse* meta-relation is a binary relation between properties, that

is *inverse* $\subseteq$ P $\times$P. Here is the axiom that define inverse relation:

($\Rightarrow$(and (class-object-component ?A ?P ?B)

(inverse ?P ?Q)(class-object ?B)) (class-object-component ?B ?Q ?A))

We assume that the inverse relation is symmetric.

The *role-hierarchy* meta-relation is a binary relation between HKS relations, that is $role-hierarchy \subseteq P \times P$. Here is the axiom that define role-hierarchy relation:

(⇔(class−object−component ?A ?U ?X))(and(role−hierarchy ?U ?V)

(class−object−component ?A ?V ?X))

We assume that the role-hierarchy relation is reflexive. Also it can be proved that role-hierarchy transitive.

Let P be the set of all properties.

Definition 5 (Class Component, Class-Object Component). A class has associated with it a collection of *components* that describe own values considered to hold for each class object of the class. Let Comp(C) be the set of components for a class C.

Let A be a class and p be a HKS relation. If (A,B) ∈ p then the triple (A, p,B) is called a *component* of class A.

A class-object has associated with it a collection of *class-object components*. Let OComp(o) be the set of class-object components of class object o. If a is a class-object of A, (A, p,B) is a component of A, b is a class-object of B, then the triple (a, p, b) can be a class-object-component of a.

Definition 6 (Root). There exist an unique special class called Root that cannot have superclasses and cannot have any components. Root can have subclasses.

Definition 7 (Knowledge Base). A Hierarchical Knowledge Base - HKB is a tuple

$$HKB = (T , C,O,P,R)$$

where T is the set of data types, C is the set of classes, O is the set of class objects and

$$R = \{subclass-of, instance-of, inverse, role-hierarchy\}$$

is the set of relations defined above.

REFERENCES

1. *** CUP, http://www.cs.princeton.edu/~appel/modern/java/CUP/
2. *** Florid, http://www.informatik.unifreiburg.de/~dbis/florid.
3. *** http://ontoknowledge.org
4. *** JAXP, http://java.sun.com/xml/jaxp/
5. *** Knowledge Interchange Format (KIF), http://logic.stanford.edu/kif/kif.html
6. *** LOG4J, http://jakarta.apache.org/, Apache Logging Services Project
7. *** OIL, http://www.ontoknowledge.org/oil
8. *** OilEd, http://oiled.man.ac.uk
9. *** Ontoedit , http://www.ontoprise.de
10. *** OntoWeb report on a comparative study of 11 ontology editors plus several other ontology tools, May, 2002 at (http://ontoweb.aifb.uni-karlsruhe.de/About/Deliverables/D13_v1-0.zip).

11. *** OWL – Web Ontology Language, http://www.w3.org/2004/OWL

12. *** Protégé , http://protégé.stanford,edu

13. *** RDF (W3C Recommendation), http://www.w3.org/RDF/

14. *** Schema Specification 1.0. Candidate recommendation, World Wide Web Consortium, Mar. 2000. See http://www.w3.org/TR/2000/CRrdfschema20000327.

15. *** The DARPA Agent Markup Language http://www.daml.org/2001/03/daml+oilindex

16. *** WebOnto, http://kmi.open.ac.uk/projects/webonto/

17. BRICKLEY, D and R, GUHA. Resource Description Framework (RDF), 2000.

18. FENSEL, D. et al : **OIL in a nutshell In: Knowledge Acquisition, Modeling, and Management**, Proceedings of the European Knowledge Acquisition Conference (EKAW2000), R. Dieng et al. (eds.), Lecture Notes in Artificial Intelligence, LNAI, SpringerVerlag, October 2000.

19. FENSEL, D. FRANK VAN HARMELEN, and I. HORROCKS. **OIL: A Standard Proposal for the Semantic Web**. Deliverable in the European IST project OnToKnowledge, 2001.

20. GIURCA, A. **A Basic Framework for Representing Ontology** , 1st National Conference on Artificial Intelligence and Digital Communication, June 2001, pp. 27-35 (hosted by Research Center in Artificial Intelligence, Craiova)

21. GIURCA, A. **A Java API for Representing Hierarchical Knowledge**, KickOff Meeting, OntoWeb European Project, SIG3 on EnterpriseStandard Ontology Environments, Crete, Greece, workshop, June 2001

22. GREGORY. J, **Story Outline for Knowledge Acquisition Tool** (Self Running Demonstrator)

23. HORROCKS, I. et al : **The Ontology Inference Layer OIL** , unpublished paper, 2001.

24. M. KIFER, G. LAUSEN, and J. WU. **Logical foundations of object oriented and framebased languages**. Journal of ACM, May 1995

25. MIKE BROWN, **Scenarios for the Knowwledge Base Video**, Interprise GmbH, German, 2001.

26. NOY, N. F., R. W. FERGERSON, M. A. MUSEN. **The knowledge model of Protégé 2000: Combining interoperability and flexibility.** 2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan les Pins, France, 2000.

27. RAY, ERIK T. **Learning XML. Sebastopol**, California: OReilly & Associates, 2001.

28. STAAB, S., A. MADCHE. **Ontology Engineering beyond the Modeling of Concepts and Relations**, http://citeseer.nj.nec.com/staab00ontology.html, 2000.