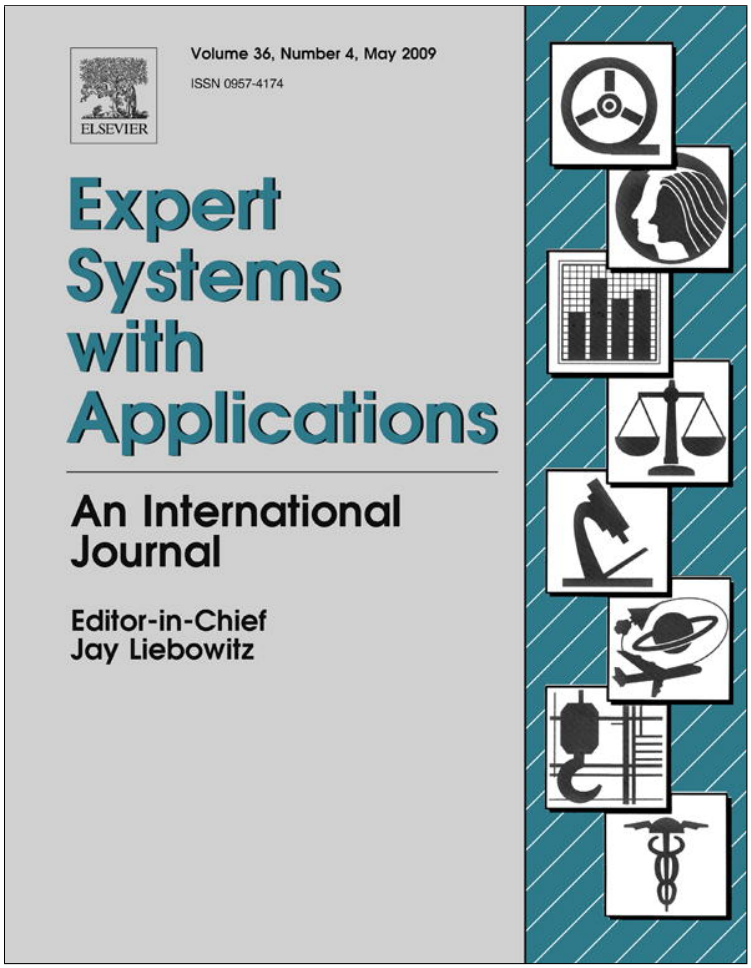

Expert Systems with Applications 36 (2009) 8185–8196

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Dynamic test generation over ontology-based knowledge representation in authoring shell

Branko Žitko *, Slavomir Stankov, Marko Rosić, Ani Grubišić

Faculty of Natural Science, Mathematics and Kinesiology, University of Split, Teslina 12, 21000 Split, Croatia

ARTICLE INFO

Keywords:
Intelligent tutoring system
Web ontology language
Dynamic quiz generation
Logic reasoning

ABSTRACT

Intelligent tutoring systems are kind of asynchronous e-learning systems designed to support and improve learning and teaching process in particular domain knowledge. An authoring shells are kind of e-learning systems that feature authoring environments for system users. Domain knowledge in such systems can be represented by using different knowledge representation specifications and presentation of tests mainly depends on the type of domain knowledge. We propose templates for dynamical generation of questions as a test over previously formalized domain knowledge. In our approach we encourage expressiveness of ontology for describing domain knowledge. Tests described in this paper entails declarative knowledge formalized by Web Ontology Language (OWL) and are realized as a dynamic quiz. By pronouncing OWL ontology as domain knowledge formalism we deal with the problem of generating tests and understanding presentation of the tests.

1. Introduction

E-learning systems are computer systems that can support classical lectures and can be applied as a supplementary instrument in learning and teaching process. Concerning the technology for delivering educational contents these systems can be categorized as asynchronous and synchronous. Intelligent tutoring systems (ITS) are kind of asynchronous e-learning systems designed to support and improve learning and teaching process in particular domain knowledge. They are knowledge oriented and: (i) have knowledge about some domain, (ii) have pedagogical knowledge about how to teach, and (iii) have knowledge about the student's ability. Besides they incorporate communication interface for interacting with the student. These characteristics are enclosed in intercommunicating modules and are known as: expert's module with domain knowledge, the student diagnostic module, the tutor module with deficiencies in knowledge and strategy for that knowledge, and finally the instructional environment with student's interface and tutorial communication (Burns & Capps, 1988).

ITSs have been often built from the outset as well as much inapproachable to the teaching experts. To bring ITS closer to the other kind of users necessitated in providing environment for the users who will set knowledge, tests and pedagogical rules in the system. An authoring shells are that kind of e-learning systems that feature authoring environments for system users. Two major reasons for the development of authoring shells are (i) lowering the cost of development and time for development and (ii) providing environment for teachers (Murray, 1998). In our approach we interpret basic structure of authoring shell as shown in Fig. 1.

It is important to highlight the difference between domain knowledge and course (Stankov, Rosić, Žitko, & Grubišić, 2008) in this class of the system. Initiative is to assemble courseware objects using particular elements taken from various domain knowledge. The structure of domain knowledge elements depends on the nature of knowledge in e-learning systems which can be declarative or procedural. Declarative knowledge nature is static and answers the question "What (is something)", while dynamic feature of procedural knowledge answers the question "How (is something done)".

Courseware, as didactically prepared learning and teaching material, has knowledge about how to teach student. Portions of domain knowledge are assembled into courseware objects and are delivered to the student in some order. Starting from this aspect existence of courseware depends on the existence of domain knowledge.

Expert's functionality is to use dedicated authoring environment to design domain knowledge. Domain knowledge can be represented by using different knowledge representation specifications, so he has to know how to formalize domain knowledge. Ideal authoring environment will provide natural language interface for domain knowledge input, and will use some kind of machine understanding techniques before saving such domain knowledge. Teacher, as course designer, has to utilize domain knowledge stored in the system to assemble courseware objects and grant

* Corresponding author.
E-mail address: branko.zitko@pmfst.hr (B. Žitko).

0957-4174/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2008.10.028

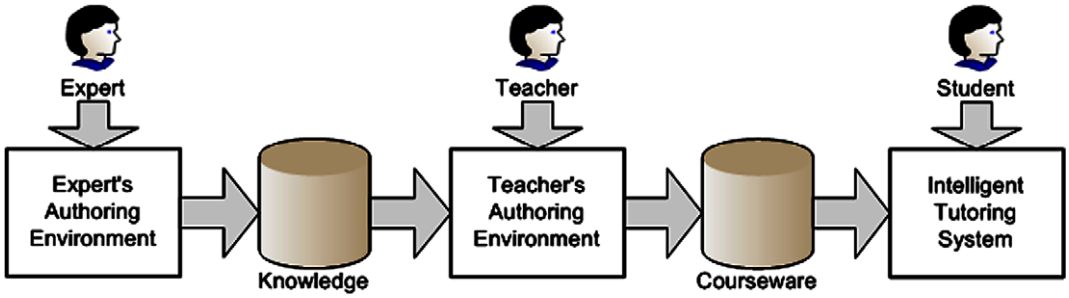

Fig. 1. Authoring shell structure (Stankov, 2003).

them hierarchical order. He is the one who decides what, how and when will student learn.

Finally, by finishing of courseware design, ITS has all data for providing learning and teaching environment to the student. Computer tutor uses different techniques for student's guidance in learning and teaching process. Student during his learning and teaching process, served by ITS, has to deal with knowledge tests which can be of different kinds. Firstly, a test depends on type of domain knowledge (declarative or procedural) and secondly, tests also have different types, like objective tests and quizzes.

Quizzes are the first to be implemented and currently the most well developed and most accepted component in the e-learning systems (Brusilovsky & Miller, 1999, 2001). Majority of these e-learning systems support authoring and delivery of quizzes made from static questions like WebCT (http://www.webct.com/), Moodle (http://www.moodle.org/). Designers of such systems made their quizzes more powerful by supporting larger variety of question types, making authoring of questions easier and supporting their management. Despite all of these upgrades of static quizzes, creating each question is still a time consuming process. Another lack is the risk of cheating during exams, since these systems usually have fixed number of static questions.

One of the solutions to these problems is providing templates with multiple choice questions where a system randomly generates instances of these questions (Santos, Simones dos Santos, Dionisio, & Duarte, 2002). Another solution is proposed in QuizPACK (Pathak & Brusilovsky, 2002) which uses dynamically parameterized quizzes that can instantiate different questions each time when quiz is accessed by the student. However, this approach evaluates knowledge for a particular programming language and is not intended to be used for other types of knowledge. There are efforts to provide templates for quizzes or questionnaires to evaluate knowledge for any domain (Quevedo-Torrero & Rodriguez-Bernal, 2007).

In this paper, we propose templates and algorithms for dynamical generation of questions over previously formalized domain knowledge in our dynamic quiz subsystem. Our proposal is outcome for over a decade of research, development and application of hypermedial authoring shell Tutor-Expert System (TEx-Sys) (Stankov, 1997) and containing quiz module that is briefly presented in Section 2. Definitely, knowledge representation plays important role as an input into automatic test generation process. In our approach we encourage expressiveness of ontology for describing domain knowledge. Tests described in this paper entails declarative knowledge formalized by Web Ontology Language (OWL) (Bechhofer et al., 2004) and are realized as a quiz (Žitko, 2005).

In Section 3, we present global architecture of dynamic quiz as a subsystem for any intelligent tutoring systems or some authoring shell whose knowledge representation involves OWL. Benefits of ontology-based knowledge representation are presented in Section 4. In Section 5, process for dynamic test generation is described in detail along with templates for dynamic tests' question and answers, decision algorithms in dynamically generated quiz as well as testing knowledge using dynamically generated quiz. Based on this idea we have build prototype system whose knowledge is formalized by OWL. We have tested this system on a chosen group of students, and finally, a result of these tests validates values of the new approach, and is showed in conclusion. Further improvement of this prototype system will be realized by applying natural language generation on OWL. This next step requires particular design of morphological lexicon as well as production rules for defining syntax of natural language sentences. Our recent research and development is directed into this field, and generally, on applied natural language processing in ITS.

2. Background

The TEx-Sys model is designed on the idea of authoring shells, as well as, cybernetic model of the system (Božičević, 1980; Wiener, 1948). It has been developed on faculty of Natural Science and Mathematics in Split, Croatia and has been used as a support for classroom activities over last ten years on 3 faculties and some primary schools in Croatia. In the period from 2001 to 2007, total of 5482 knowledge tests were solved by 1302 students.

Promotion of Information and Communication Technology, new discoveries on the ITS field as well as the results of TEx-Sys's usage brought up 3 versions of TEx-Sys model.

First version of TEx-Sys model (in period from 1992 to 2001) is on-site application relaying on relation database. Distributed Tutor-Expert System (DTEx-Sys) (in period from 1999 to 2003) (Rosić, 2000), as the second version, uses dynamic Web page generation to expand boundaries out of classroom and to provide its service to anyone, anywhere and anytime. Third version, Extended Tutor-Expert System (xTEx-Sys) (in period from 2003 to 2005) (Stankov, 2003) is also Web application but with its architecture based on Web service (Rosić, Glavinić, & Žitko, 2004).

All of these versions have important common characteristics. Semantic network with frames is used for declarative knowledge representation and tests are in the form of the quizzes which are dynamically generated (Stankov, Rosić, & Glavinić, 2001). What is not common is that courseware, as important data entity, is introduced in xTEx-Sys and we can say that xTEx-Sys is Web-based authoring shell that provides environment for the teacher and student to participate. Also, expert is one of the system's user roles who designs domain knowledge and has to be prepared before doing so. He has to formalize domain knowledge using semantic network with frames and input it into xTEx-Sys knowledge base. Usually teacher has expert's and teacher's role in the system, but before teacher can design a courseware, he has to build knowledge base or use some or all elements of previously built knowledge bases in his courseware.

Courseware in xTEx-Sys organizes course objects, that deliver domain knowledge (learning objects), and objects for delivering

test (testing objects) to the student. These objects are "leafs" of tree-like hierarchical courseware structure. This Sharable Content Object Reference Model (SCORM)-like (www.adlnet.org) structure, unlike SCORM Content Aggregation Model (Fig. 2), represent learning objects as a collection of semantic network nodes from different knowledge bases and has testing objects which are realized as dynamical quizzes based on elements from semantic network with frames.

These testing objects generate quiz questions by using 12 hard-coded question and answer templates which are grouped by 3 heaviness categories with 4 questions and answers templates in each. Dynamic quiz in xTEx-Sys is adaptable because it depends on student answers on a series of questions. If student doesn't know the answer from one series of questions then next series will contain easier questions, and vice versa.

Interviews with the students that have been using xTEx-Sys system were very encouraging but they have pointed out some important flaws. The most important is that even so quiz initiation brings up every time different questions, text of the questions and answers are highly formalized and dependent on semantic network with frames (Fig. 3).

This can be very confusing and make boundaries to the first time users of the system, which can be overridden by introducing student to the formalism of semantic network with frames.

Dealing with this problem, it makes us select another declarative knowledge representation and implement it in TEx-Sys model. Web Ontology Language (OWL) is our choice because of its nature to provide graphical viewpoint of described domain knowledge, as well as to develop expressiveness realized by logical expression constructors. Our work in making TEx-Sys system better and more understandable to the student was to expand the way the questions and answer choices are generated by enabling design of the question/answer templates out of the system's code.

To establish such functionality, TEx-Sys system has been intercommunicating with subsystem which provides public interface with methods for passing knowledge base formalized by OWL, as well as, methods for interpreting XML question/answer templates. Output of such subsystem consists of the interpretation of question and answers. In following is given basic overview of subsystem components.

3. Global architecture of dynamic quiz subsystem

Dynamic quiz subsystem whose architecture is showed on Fig. 4 is actually a part of ITS, and can be considered as a component with interface to student's module in ITS. This subsystem has published interface with simple methods for setting domain knowledge and question/answer templates, as well as method for getting interpreted question with answer.

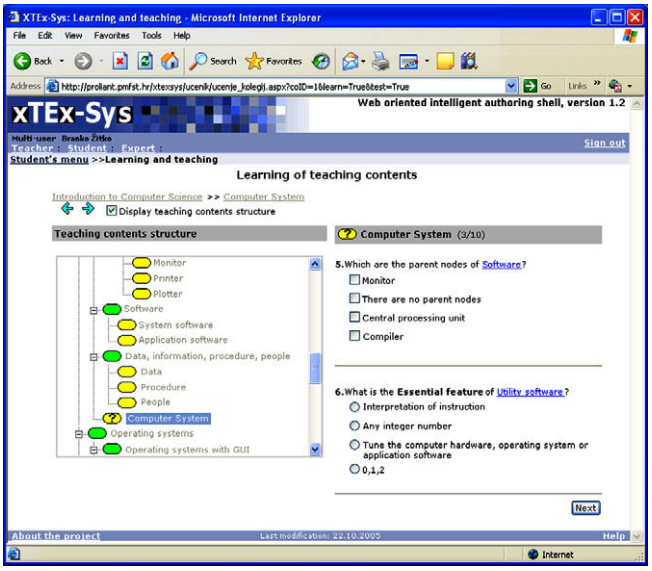

Fig. 3. Quiz in xTEx-Sys.

Basic idea is to apply domain knowledge on some question/answer template and use reasoning to generate question and answers.

For generating quiz based on question/answer template over some given domain knowledge, system has to transfer domain knowledge represented by OWL language into DIG format (description logic implementation group) (http://dl.kr.org/dig) and post it to external description logic (DL) (Donini, Lenzerini, Nardi, & Schaerf, 1996; Nardi & Brachman, 2003) reasoner. Interpretation of question/answer template interacts with DL reasoner posting ask expressions and requesting tell expressions whose interpretation conjoins question/answer template and construct question as well as set of at least one correct and few incorrect answers.

Core function of dynamic quiz subsystem is to interpret question and answers for given template and knowledge representation.

Modules of ITS are responsible for overall execution and flow of tests based on dynamic quiz in ITS.

Hereinafter are described data and processes inside dynamic quiz subsystem; therefore first of all is briefly described knowledge representation by OWL following by specification of question/answer templates in addition to explanation of algorithms and workflows involved in process of testing student's knowledge. Again we set off OWL ontology as a knowledge representation in dynamic test generation since data structures and algorithms included in test generation are highly dependent on it.

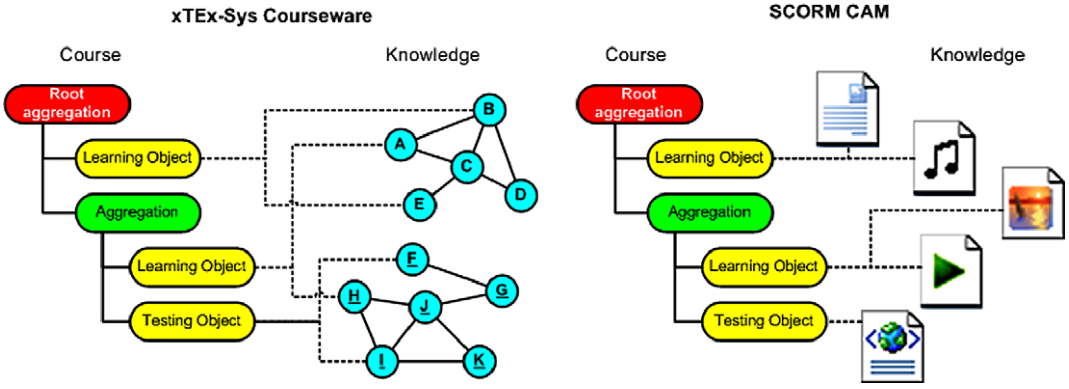

Fig. 2. Difference between courseware in xTEx-Sys and SCORM.

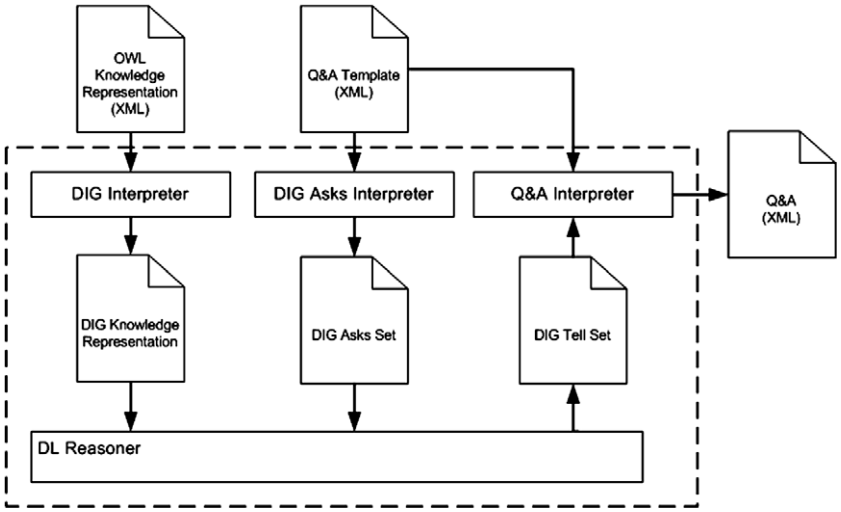

Fig. 4. Dynamic quiz subsystem.

4. Benefits of ontology-based knowledge representation

Our specific expertise and experience gained during usage of TEx-Sys model has compelled us to change the way that domain knowledge is represented, mainly because semantic network with frames did not satisfy our need for greater expressiveness of declarative knowledge. OWL Ontology is our new choice primarily for its machine processability, while still being natural enough. Secondarily, OWL Ontology possesses semantic clarity to annotations and can be easily integrated on WWW.

OWL Ontology as knowledge representation has foundation in description logic. More accurately OWL DL (Bechhofer et al., 2004) sublanguage constructors contain equivalents in SHOIN(D) axioms and facts. Such axioms and facts are written in RDF/XML external format giving stronger meaning and description for resources on the Web, supporting Semantic Web realization.

Naturally, SHOIN(D) is complete; meaning that all inferences are computed in finite time. SHOIN(D) as description logic language has some constructors that can be described using semantic network representation. For example, if some class B is a subclass of another class A then in semantic network it is represented by two generic nodes A and B connected by A_KIND_OF link from node B to node A. If there is an individual X of class A then in semantic network is represented by connecting generic node A using link INSTANCE_OF with instance node X.

Naturally, semantic network describes some domain using nodes and their connections, and collection of those connections provides a logical meaning using AND operator.

Axioms in SHOIN(D) can describe domain concepts more precisely using logical operators AND, OR and NOT between class descriptions. Other refinement is given using restriction on properties telling, for example, that some class which is in domain of some property can have all values, some values or exactly one value from class described in a range of property, or can have at least, at most or equal number of instances for property value. Such rich language makes firmer constraints for describing things precisely in some domain of interest. Properties in SHOIN(D) provide schema for frame-like language. Object properties are mathematical relation, and as such, they can be functional, transitive, and symmetric or can be inversed.

Expressions in OWL DL sublanguage describe things by using axioms. Description of class limits instantiation of its individuals. Simple example of domain knowledge on the course of computer system and their relations is used in this paper in for purpose of describing dynamic quiz generation algorithms. In Fig. 5 our representation of domain knowledge in OWL abstract syntax (Peter, Patel-Schneider, Hayes, Horrocks, & van Harmelen, 2003) format is given.

For example, hardware is defined as computer system component that has central devices, input devices and output devices. So #Hardware is a subclass of #Computer_system_component and is described as a union of #Central_device, #Input_device and #Output_device. Another example describes #Primary_storage as #Storage which is volatile.

If something is an instance of #Storage and is volatile then inference will tell that it is also #Primary_storage.

Inference process in this example is executed on reasoner for description logic. The most known description logic reasoners provide DIG interface for setting knowledge and for ask and tell operations. DIG interface is effectively represented as an XML Schema for a DL concept language along with ask/tell functionality.

Different reasoners have different capabilities, and in order to cope it, along with information regarding their identification, a reasoner should also supply details of the language which it supports. It is assumed that all reasoners will support primitive concepts and roles. For our purposes we have chosen FaCT++ (Horrocks, 1998, 1999) version 1.1.4 which is a SROIQ(D) description logic reasoner supporting DIG 1.1 (Dickinson, 2004) (http://dl.kr.org/dig) interface that includes the standard boolean concept operators (and, or, not), universal and existential restrictions, cardinality constraints, a role hierarchy, inverse roles, the one-of construct and concrete domains.

Concept language of FaCT++ provides:

- Primitive concepts with predefined top and bottom concept as well as syntax to make simple concepts.
- Boolean operators for defining concepts using conjunction, disjunction and negation of other concepts.
- Property restrictions to make constraints on concepts by defining restrictions on role values.
- Concrete domain expressions to make constraints on concepts by defining restrictions on attributes values.
- Property expressions for defining roles and attributes and their inverses.
- Individuals for describing individuals by their name.

A tell request made to particular knowledge base additionally describes facts using:

OWL abstract syntax

```
Class(#Computer_system complete
intersectionOf(restriction(#has_computer_system_function
allValuesFrom(#Computer_system_function))
restriction(#has_computer_system_component
allValuesFrom(#Computer_system_component))))
Class(#Computer_system_component partial)
Class(#Hardware complete #Computer_system_component unionOf(#Central_device
#Output_device #Input_device))
Class(#Central_device complete #Hardware unionOf(#Memory #Computer))
Class(#Computer partial #Central_device restriction(#has_function
allValuesFrom(#Processing)))
Class(#Memory partial #Central_device restriction(#has_function
allValuesFrom(#Storage)))
Class(#Main_memory partial #Memory)
Class(#RAM partial restriction(#is_volatile value("true"^^xsd:boolean))
#Main_memory)
Class(#ROM partial restriction(#is_volatile value("false"^^xsd:boolean))
#Main_memory)
Class(#Secondary_memory partial #Memory)
Class(#Floppy_disk partial #Secondary_memory)
Class(#Input_device partial #Hardware restriction(#has_function
allValuesFrom(#Input)))
Class(#Keyboard partial #Input_device)
Class(#Output_device partial #Hardware restriction(#has_function
allValuesFrom(#Output)))
Class(#Monitor partial #Output_device)
Class(#Printer partial #Output_device)
Class(#Speaker partial #Output_device)
Class(#Software partial #Computer_system_component)
Class(#Computer_system_function partial)
Class(#Input partial #Computer_system_function)
Class(#Output partial #Computer_system_function)
Class(#Processing partial #Computer_system_function)
Class(#Storage partial #Computer_system_function)
Class(#Primary_storage partial #Storage restriction(#is_volatile
value("true"^^xsd:boolean")))
Class(#Auxiliary_Storage partial restriction(#is_volatile
value("false"^^xsd:boolean")) #Storage)
```

```
Individual(#HP_Optical_Mouse type(#Mouse))
Individual(#Canon_Laser_Printer type(#Printer))
Individual(#Noname_DDR_RAM_512MB type(#RAM))
```

```
ObjectProperty(#has_computer_system_function domain(#Computer_system)
range(#Computer_system_function) Transitive)
ObjectProperty(#has_computer_system_component domain(#Computer_system)
range(#Computer_system_component) Transitive)
ObjectProperty(#has_function domain(#Computer_system_component)
range(#Computer_system_function))
ObjectProperty(#manage_data domain(#Computer_system_function) range(#Data)
Functional)
ObjectProperty(#accept_data super(#manage_data) domain(#Input))
ObjectProperty(#process_data super(#manage_data) domain(#Processing))
ObjectProperty(#produce_data super(#manage_data) domain(#Output))
ObjectProperty(#store_data super(#manage_data) domain(#Storage))
ObjectProperty(#is_part_of inverseOf(#has_part) Functional)
ObjectProperty(#has_part inverseOf(#is_part_of) Transitive)
```

```
DatatypeProperty(#is_volatile Functional domain(#Memory) range(xsd:boolean))
```

Fig. 5. Example of domain knowledge in OWL abstract syntax.

- Primitive concept introduction for defining concepts, roles, attributes and individuals.
- Concept axioms for defining equivalent or disjoint concepts.
- Role axioms for defining equivalent or disjoint roles and attributes, as well as their domains and ranges.
- Individual axioms for defining values for types individual.

Transformation from RDF/XML syntax of OWL ontology to DIG XML syntax is relatively easy, knowing that classes, object properties, data type properties and individuals in OWL correspond to concepts, roles, attributes and individuals in DIG. In Table 1 example of transforming OWL expression that describes class #Hardware is given.

Querying reasoner is provided through ask request which itself consists of a number of ask statements. Each ask statement is uniquely identified, and in such way, allows presentation of multiple queries in one request. Moreover, each ask request must also have an attribute that identifies the knowledge base that the queries.

Table 1
Transformation from OWL syntax to DIG syntax.

RDF/XML	DIG

`<owl:Class rdf:ID="Computer_system">`
`  <owl:equivalentClass>`
`  <owl:Class>`
`  <owl:intersectionOf rdf:parseType="Collection">`
`  <owl:Restriction>`
`  <owl:allValuesFrom rdf:resource="#Computer_system_function"/>`
`  <owl:onProperty>`
`  <owl:TransitiveProperty rdf:ID="has_computer_system_function"/>`
`  </owl:onProperty>`
`  </owl:Restriction>`
`  <owl:Restriction>`
`  <owl:allValuesFrom>`
`  <owl:Class rdf:ID="Computer_system_component"/>`
`  </owl:allValuesFrom>`
`  <owl:onProperty>`
`  <owl:TransitiveProperty rdf:ID="has_computer_system_component"/>`
`  </owl:onProperty>`
`  </owl:Restriction>`
`  </owl:intersectionOf>`
`  </owl:Class>`
`  </owl:equivalentClass>`
`  </owl:Class>`
	`<defconcept name="Computer_system"/>`
	`  <equalc>`
	`  <catom name="Computer_system"/>`
	`  <and>`
	`  <all>`
	`  <ratom name="has_computer_system_function"/>`
	`  <catom name="Computer_system_function"/>`
	`  </all>`
	`  <all>`
	`  <ratom name="has_computer_system_component"/>`
	`  <catom name="Computer_system_component"/>`
	`  </all>`
	`  </and>`
	`  </equalc>`

A ask request syntax made to particular knowledge can be grouped by

- Primitive concept retrieval for listing all concept, role or individual names.
- Satisfiability for discovering relation between concepts.
- Concept hierarchy for getting concept position in hierarchy.
- Role hierarchy for getting role position in hierarchy.
- Individual queries for discovering all individuals of some concept or for retrieving concepts that individual belong to, as well as, getting values of individual roles.

The response to an ask request must contain, in its body, a responses element, which itself consists of a number of responses – one for each query in the ask request. Each particular response must have an identifying attribute which corresponds to the identifier of a submitted query.

Possible responses to an ask request can be classified by

- Boolean responses telling whether answer is true or false.
- Concept set of synonyms (names of equivalent concepts) satisfying the request.
- Role set of synonyms (names of equivalent roles) satisfying the request.
- Individual set satisfying the request.
- Individual pair set containing pairs of individual in relation to some other or the same individual.
- Values containing value for individual's attribute.

Example of querying reasoner for discovering most specifying concept of personal computer is given in Table 2.

However even so OWL ontology can describe concepts using restrictions over data type properties, transformation of such restricted descriptions is not provided in DIG 1.1 interface, so we have to deal with it by implementing reasoning support over data type properties in our test application.

5. Dynamic test generation

Domain knowledge formalized by OWL is one kind of input data in our dynamic quiz subsystem. The other is the set of templates which describes how questions will be generated and how they are going to be presented. For a specific domain knowledge some templates can be used for generating questions, therefore there is a need for algorithms to decide if specific template can be interpreted. Finally, when dynamic quiz subsystem has created a set of interpretable templates and if this set has adequate number of templates, a test can be started. Furthermore in this section, we have specified all necessary data structures for designing question and answers in the test, as well as decision algorithms in dynamic quiz subsystem. In the end, we describe run-time process of testing and evaluating student's test's results.

5.1. Templates for dynamic test's question and answers

OWL ontology is a collection of facts and axioms that represents a logical theory in which a collection of entailed facts and axioms are true. Facts and axioms can be interpreted as a collection of sentences. Therefore, query interpretation requests for entailed sentences that satisfy query sentence schema, and uses bindings to variables in that sentence as specifying query answers (Fikes, Hayes, & Horrocks, 2005). For example, query "What is hardware?" and answer "Hardware is type of computer system component" will have following query and answer patterns as well as binding variables.

Table 2
Example of DIG query and answer.

DIG Query	DIG Answer
<asks uri="http://dl.kr.org/dig/FaCTpp- 000_24-50-0_25-3-2007"> <types id="Query1"> <individual name=" #Personal_computer"/> </types> </asks>	<responses> <conceptSet id="Query1"> <synonyms> <catom name="#Computer_system"/> </synonyms> </conceptSet> </responses>

```
Query: What is hardware?

Query pattern: {(type hardware ?X)}
Must-bind variables: (?X)
Answer pattern: {(hardware type ?X)}
Answer: Hardware is type of computer system component.
Answer pattern instance: (hardware type computer system component)
```

Query pattern is an ordered set of triples of form <property> <subject> <object> where any item in a triple can be a variable (Haarslev, Möller, & Wessel, 2004). Each triple in a set can be true for some variable, and answer exists if conjunction of all of query pattern triples is true.

It is designed for answering queries of the form "What URI references and literals from the answer KB and OWL denote objects that make the query pattern true" or, "Is the query pattern true in the answer KB". In the first form query pattern specifies must-bind variables which will be used in answer interpretation. Second form has no variables bound to the query pattern. These patterns are used in the following for defining question/answer templates.

```
Example: Is optical mouse computer function?

Type: YesNo
Question pattern: Is ?i ?c
Random Bind: (?i Individual)(?c Class)
Answer Patterns:
Correct Answer Pattern: {(?x type ?y)(?y subsumes ?c)}
Must Bind: (?x)
Correct Answer Pattern Instance: Yes
Correct if: in(?i, ?x)
Incorrect Answer Pattern Instance: No
```

In this example question is asking to answer Yes if some individual is belonging to some class. Type of this question is Yes/No meaning that there are only two possible answers and only one of them is correct. Question pattern has two random binding variables ?i and ?c. Random binding means to randomly choose one element from set of all elements. In this example ?i is going to be randomly chosen individual and ?c is going to be randomly chosen class.

By correct answer, pattern is defined by using the way of selecting all individuals described by variable ?x which are belonging to some class ?y which is subsumed by randomly chosen class ?c.

If randomly chosen variable ?i is in ?x then answer would be correct and Yes, otherwise No will be displayed on screen.

Another example uses binding variables for defining query pattern for the multiple choice question:

```
Type: MultiChoice

MinCorrect: 1
MaxCorrect: 2
Total: 4
Example: Select mouse!
Question pattern: Select ?c
Random Bind: {(?c Class)}
Answer Patterns:
Correct Answer Pattern: {(?x type ?y)(?y subsumes ?c)}
Must Bind: (?x)
Correct Answer Pattern Instance: None
Correct if: countEqual(?x, 0)
Correct Answer Pattern Instance: ?x
Correct if: countGreater(?x, 0)
Incorrect Answer Pattern: {(?x type ?y)(?y subsumes complement((?c))}
Must Bind: (?x)
Incorrect Answer Pattern Instance: None
Incorrect if: countEqual(?x,0)
Incorrect Answer Pattern Instance: ?x
Incorrect if: countGreater(?x, 0)
```

In this type of question there can be 1 or 2 correct answers where total number of answers is 4. Question pattern has one randomly binded variable ?c which represents some class. Answer on this question would be some individual or individuals which are belonging to that randomly chosen class ?c.

Correct answer pattern binds all individuals represented by variable ?x which belong to class ?y subsumed by class ?c. If there is no such individual then answer would be None, otherwise correct answer pattern instance would be one or more individuals in ?x. Incorrect answer pattern is correspondingly built.

Question/answer template consists of query pattern and answer patterns. This abstract syntax is expressed in XML format as showed in Fig. 6.

This template and domain knowledge represented by OWL are inputs for subsystem. Output will be slightly formalized question and set of answers where one or more of them are correct.

QATemplate element defines this question as multiple choice having 4 answers where at least 1 and at most 2 answers are correct. Question/answer templates are also categorized as easy, medium and hard. This categorization is used for adaptation of quiz during quiz execution. Text of this template will always start with "Select" following by name of randomly chosen class, for example "Select input device" where input device is a name of class. This template consists of correct and incorrect answer specification. CorrectAnswer element describes pattern for correct answer as well as binding variables and presentation of correct answer.

```
<QATemplate type="MultipleChoice" minCorrect="1" maxCorrect="1"
category="Heavy">
  <QuestionPattern>
    <TextElement type="text">Select</TextElement>
    <TextElement type="variable">c</TextElement>
  </QuestionPattern>
  <RandomBind variable="c" source="Class"/>
  <AnswerPatterns>
    <Answer correct="true">
      <AnswerPattern>
        <AnswerAtom>
          <Subject><Variable>x</Variable></Subject>
          <Predicate><DIGAsk>type</DIGAsk></Predicate>
          <Object><Variable>y</Variable></Object>
        </AnswerAtom>
        <AnswerAtom>
          <Subject><Variable>y</Variable></Subject>
          <Predicate><DIGAsk>subsume</DIGAsk></Predicate>
          <Object><Variable>c</Variable></Object>
        </AnswerAtom>
      </AnswerPattern>
      <MustBind variable="x"/>
      <AnswerInstancePattern>
      <TextElement type="text">None</TextElement>
        <Condition>
          <Count operator="equal"> <Variable>x</Variable>
            <Constant>0</Constant>
          </Count>
        </Condition>
    </AnswerInstancePattern>
    <AnswerInstancePattern>
      <TextElement type="variable">x</TextElement>
      <Condition>
        <Count operator="greater">
          <Variable>x</Variable>
          <Constant>0</Constant>
        </Count>
      </Condition>
    </AnswerInstancePattern>
    </Answer>
  </AnswerPatterns>
</QATemplate>
```

Fig. 6. Question/answer template in XML format.

Pattern of correct answer is a set of answer atoms where each atom has subject, predicate and object element. Each of these elements can be a constant text, a variable value or some DIG element for ask query. In a case of incorrect answer, elements that interpret answer consists of description how text of answer is interpreted as well as rules that match such interpretation. Rule elements usually compare some binded variable with condition, such as CountEqual, CountGreater and similar. XML Schema for specifying question/answer template is given in Fig. 7.

By using this specification, several question/answer templates are already built and their simplified syntax as well as heaviness category is displayed in Table 3.

Question/answer template from Fig. 5 is a multiple choice and has 4 answers where one or two of them can be incorrect. In order for this template to be interpreted there has to be at least one individual belonging to class or some of its subclasses. Algorithm for deciding if template is interpretable mainly depends on question type, number of answer and on binding variables. If domain knowledge does not count enough classes, individuals or properties, then question/answer template will probably not be interpreted, so first task for algorithm is to count elements of domain knowledge and count how many of them some template will require.

Select input device:

1. None.
2. HP Optical mouse.
3. Canon Laser printer.
4. DDR RAM 512MB.

For example, question/answer template from Fig. 5 will have one class name in question and 3 or 4 individual names in answer. At least one answer is correct so algorithm has to find conditions where answers are correct. By template from Fig. 5, first correct answer instance specifies that answer "None" is correct if there is no individual directly or indirectly subsumed by class or some of its subclass from question. Second correct answer instance specifies that individual name is correct answer if that individual is instance of some class that is subsumed by the question class. By analyzing

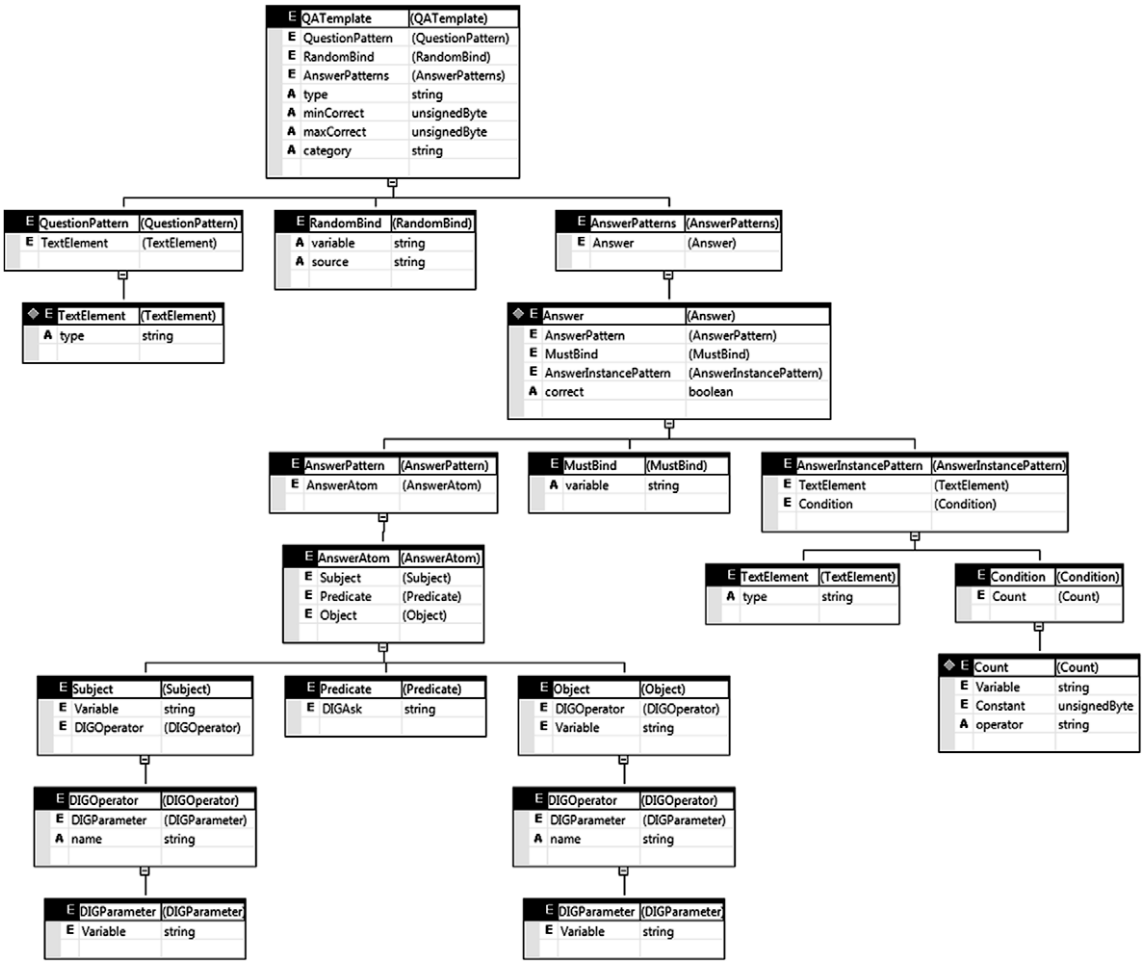

Fig. 7. XML Schema for question/answers template.

Table 3
Question/answer templates simplified description.

	Question	Question type and answer type	Heaviness category
1	What is [Class]?	Multiple choice [ClassDescriptions]	Heavy
2	What is [Individual]?	Multiple choice [Class]	Heavy
3	Who has [DatatypeProperty] [Value]	Multiple choice [Individual]	Heavy
4	Select [Class]?	Multiple choice [Individual]	Heavy
5	Does [Class] [ObjectProperty] [Class]?	True/false	Easy
6	Does [Individual] [ObjectProperty] [Individual]?	True/false	Easy
7	Does [Individual] [DatatypeProperty] value?	True/false	Easy
8	Is [Individual] [Class]?	True/false	Easy
9	Is [Class] [Class]?	True/false	Easy
10	Does [Individual] has [DatatypeProperty]	True/false	Easy
11	What is [DatatypeProperty] of [Individual]?	Multiple choice [Value]	Mediate
12	Who has [ObjectProperty] [Individual]	Multiple choice [Individual]	Mediate
13	What is [DatatypeProperty] of [Individual]	Multiple choice [Value]	Mediate
14	What is [ObjectProperty] of [Individual]	Multiple choice [Individual]	Mediate

this conditions algorithm finds if question and template answer is interpretable. In example, Optical mouse is correct answer because #Optical_mouse is individual belonging to the class #Input_device.

These templates as well as domain knowledge formalized by OWL are two inputs for dynamic quiz subsystem. Before dynamic quiz subsystem runs generation of questions and answers, he has to decide if quiz is generable. In following those two processes are described; determination if quiz is generable and generation of quiz.

5.2. Decision algorithms in dynamically generated quiz

Question/answer template contains data ready to be processed by algorithms in dynamically generated quiz. Some of these templates cannot be interpreted for particular domain knowledge since domain knowledge does not have sufficient number of elements or elements in relation for question to be generated. If there are not enough interpretable templates, quiz will not be executed.

Process of deciding if some quiz is executable is divided into following steps:

1. making sets of question/answer templates that can be interpreted and
2. deciding if quiz can be run by counting interpretable templates for each heaviness category and comparing those numbers with planned number of questions in quiz.

According to steps of this process we have developed two algorithms:

1. template algorithm for determining if question/answer template is interpretable and
2. deciding algorithm for determining if there are enough interpretable templates for quiz to be executed.

First dataset inputted in process of generating questions and answer choices is a domain knowledge formalized by OWL. Other dataset is a set of all available question/answer templates from which is necessary to filter those templates that are not interpretable, i.e. there is not enough elements in domain knowledge that have to be binded to template variables.

Process of filtering templates takes each template from all available templates and marks it as interpretable or not interpretable.

Template algorithm for deciding if specific template is interpretable takes in count all elements of domain knowledge as well as all variables and their relationship with other variables and also with question pattern, correct and incorrect answer patterns.

Firstly template algorithm takes question pattern and for each variable seeks knowledge base for all elements that have to be binded to such variable.

For example, in pattern:

```
MinCorrect: 1

MaxCorrect: 2
Total: 4
Question pattern: Select ?c
Random Bind: {(?c Class)}
```

there is variable ?c which will be some class from domain knowledge. If there is no classes in domain knowledge then template algorithm stops, and template is not interpretable.

In the next step, template algorithm remembers the count of minimal, maximal and total number of answers in template.

In template from example there is 1 correct answer pattern and 1 incorrect answer pattern.

Algorithm for each answer pattern counts how many of them are executable for each instance of variable from question template and in this example, it is the class ?c.

```
Correct Answer Pattern: {(?x type ?y)(?y subsumes
?c)}

Must Bind: (?x)
Correct Answer Pattern Instance: None
Correct if: countEqual(?x, 0)
Correct Answer Pattern Instance: ?x
Correct if: countGreater(?x, 0)
Incorrect Answer Pattern: {(?x type ?y)(?y subsumes
complement((?c))}
Must Bind: (?x)
Incorrect Answer Pattern Instance: None
Incorrect if: countEqual(?x,0)
Incorrect Answer Pattern Instance: ?x
Incorrect if: countGreater(?x, 0)
```

First correct answer pattern has two atoms (?x type ?y)(?y subsumes ?c) and algorithm creates forest with nodes labeled as variables or constants and edges are DIG reasoner's ask statements. The root of the forest trees are constants or variables from question patterns and leafs are usually must-bind variables from answer pattern. Fig. 8 gives graphical view of atoms from correct answer pattern.

Template algorithm makes level order traversal of the tree and visits every node following the edges. By visiting node reasoner is queried and its response set makes an input for next reasoner query. In this example there are two kind of queries, first to select all classes that are subsumed by class in variable ?c and store them in variable ?y. Then for each subsumed class in ?y reasoner gets all instances of ?y and sets variable ?x.

Question instance pattern is correct if the count of elements in some variable compared to specific number has satisfied specific condition. The first correct answer pattern instance from example is correct if the count of elements in variable ?x is equal to 0, meaning there are no individuals in class ?c or in some of classes from ?y subsumed by a class ?c, so the correct answer would be "None".

In this example, there are two correct answer instances for this correct answer pattern. First one tells that answer is correct if there are no instances of subsumed classes of ?c.

If it is true then template algorithm's counter of correct answer instances is increased. Analogue, the counter of incorrect answer instances is increased if incorrect answer instance is true for the same variable ?c.

Now those counters are compared with numbers for minimal and maximal correct answers as well as with the number of total answers. If counter of correct answers is less than the number for minimal correct answers or sum of counters of correct and incorrect answers is less than the total number of answers, then template is not interpretable for selected domain knowledge.

Another decision algorithm tries to answer on a question if quiz is executable. Main idea of this simple algorithm is to determine if there are enough interpretable templates from each heaviness category. Number of templates of each heaviness category has to be greater or equal then requested number of questions in quiz. If this is so, then quiz can be executed.

Number of questions in a quiz is initially set by teacher during development of test object in courseware. This phase and execution of a quiz, as well as presenting quiz results is described in the following section.

5.3. Testing knowledge using dynamically generated quiz

Decision algorithms are basic algorithms in a process of dynamic quiz generation. Their main task is to determine if quiz is executable for given domain knowledge sample and involved in process of dynamic quiz generation. Dynamic quiz generation has three phases:

1. quiz initialization,
2. quiz execution and
3. presentation of results.

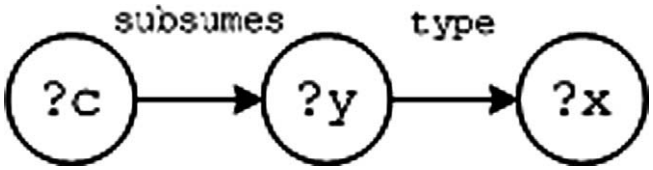

Fig. 8. Answer atoms.

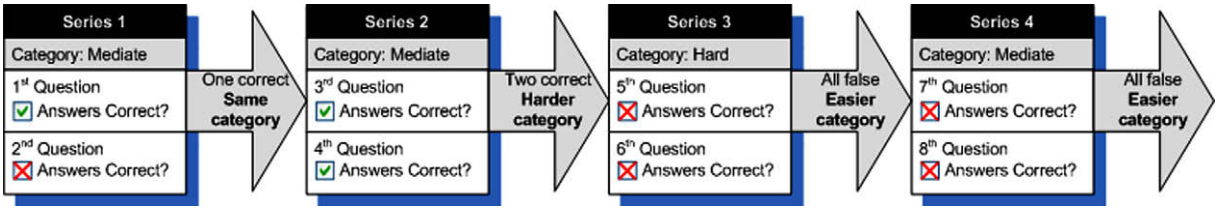

Fig. 9. Category shifting during dynamic quiz generation (Stankov, Žitko, & Grubišić, 2005).

First phase occurs during designing test object by the teacher. Initialization of quiz is made by naming test object, selecting domain knowledge that will be processed in quiz execution and selecting a number of questions in the quiz. After that system uses decision algorithms for determining whether the quiz is executable. If it is executable then testing object is stored in a courseware and is ready to be executed. Execution of quiz is initialized by the student by selecting test object in the courseware. Initialization of execution uses decision algorithms to make subset of interpretable templates for each heaviness category. Quiz execution generates two pairs of questions from the same heaviness category and shifts category for generation of next two questions depending on student's response. Shifting of the categories is shown in Fig. 9.

After last series of question, total score for each question is summarized towards calculating final mark according to the relation between accomplished points and the maximal possible points. Calculated mark varies from unsatisfying to excellent. Presenting the result of the test not only involves a display of the final mark; but also it gives back a set of all solutions of the answered questions as well as a question category sequence. Student can actually see where he or she has done wrong and afterwards choose concept or relation to see exactly where, how and why he or she had made a mistake.

6. Conclusion and further work

Our aim in this paper is to justify the way that tests can be generated employing formalized domain knowledge as well as an ability to design the elements of dynamically generated test. By pronouncing OWL ontology as domain knowledge formalism we deal with the problem of understanding presentation of the tests.

In order to confirm our assumptions about benefits of ontological knowledge representation as well as an appliance of dynamic quiz subsystem with custom templates, we have implemented experimental prototype of system including learning and testing module. Surface representation of knowledge in learning module is realized using quasi-sentences containing hyperlinks to other quasi-sentences, unlike TEx-Sys model where semantic network is represented graphically. Testing module of experimental prototype of the system is communicating with dynamic quiz system which contains predefined QA templates. A group of students from college course "Introduction to Computer Science" were learning and testing on the DTEx-Sys, and week later on experimental prototype of the system. The survey results made us to conclude that knowledge representation using OWL formalism was well and even better accepted then semantic network. Nevertheless this means, domain knowledge represented to the student, and previously made by domain expert and afterwards arranged by the teacher, is much more complex but still in the form of formalized language that can be transcendental to the users. However, questions are still grammatically incorrect and we are now highly motivated to represent knowledge and questions in natural language text. If we want to break this gap, we have to interpret formalized language to the natural language. Application of natural language processing is the next step in making this a kind of ITS more acceptable to all users, especially students.

Generating natural language sentences from formalized language is a process that can be divided in two phases. First phase requires morphological lexicon which would be capable of retrieving all word forms and their grammatical categorization.

In the second phase, syntactical rules for collecting and ordering all word forms, retrieved from lexicon, designs natural language sentences, or questions in case of quiz.

Setting natural language generator layer over formalized domain knowledge will make large impact on students, as well as, their acceptance of this kind of ITS.

References

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., et al. (2004). In M. Dean, & G. Schreiber (Eds.), *OWL web ontology language reference, W3C recommendation, 10 February 2004*. <http://www.w3.org/TR/owl-ref>.

Božičević, J. (1980). *Fundamentals of automatic control, Part I – System approach and control* (10th ed.). Zagreb: Školska knjiga (in Croatian).

Brusilovsky, P., Miller, P. (1999). Web-based testing for distance education. In P. De Bra & J. Leggett (Eds.), *Proceedings of WebNet'99, World Conference of the WWW and Internet* (pp. 149–154). AACE: Honolulu, HI, October 24–30.

Brusilovsky, P., & Miller, P. (2001). Course delivery systems for the virtual university. In T. Tschang & T. Della Senta (Eds.), *Access to Knowledge: New Information Technologies and the Emergence of the Virtual University* (pp. 167–206). Amsterdam: Elsevier Science.

Burns, H. L., & Capps, C. G. (1988). Foundations of Intelligent tutoring systems: An introduction. In M. C. Polson & J. J. Richardson (Eds.), *Foundations of intelligent tutoring systems* (pp. 1–18). Lawrence Erlbaum Associates Publishers.

Dickinson, I. (2004). *Implementation experience with the DIG 1.1 specification*. Technical report, Semantic and Adaptive Systems Dept, HP Labs Bristol.

Donini, F. M., Lenzerini, M., Nardi, D., & Schaerf, A. (1996). Reasoning in description logics. In G. Brewka (Ed.), *Foundation of knowledge representation*. CSLI Publication, Cambridge University Press.

Fikes, R., Hayes, P., & Horrocks, I. (2005). OWL-QL – A language for deductive query answering on the semantic web. *Journal of Web Semantics, 2*(1), 19–29.

Haarslev, V., Möller, R., & Wessel, M. (2004). Querying the semantic web with Racer + nRQL. In *Proceedings of the workshop on description logics 2004 (ADL 2004), University of Ulm, Germany, September 24th, 2004*.

Horrocks, I. (1998). Using an expressive description logic: FaCT or fiction? In *Proceedings of the 6th international conference on principles of knowledge representation and reasoning (KR'98)* (pp. 636–647).

Horrocks, I. (1999). FaCT and iFaCT. In *Proceedings of the 1999 description logic workshop (DL'99)* (pp. 133–135).

Murray, T. (1998). Authoring knowledge-based tutors: Tools for content, instructional strategy, student model, and interface design. *Journal of the Learning Sciences, 7*(1), 5–64.

Nardi, D., & Brachman, R. J. (2003). An introduction to description logics D. In F. Baader, D. L. McGuinness, D. Nardi, & P. F. Patel-Schneider (Eds.), *The description logic handbook: Theory, implementation, and applications* (pp. 5–45). Cambridge University Press.

Pathak, S., & Brusilovsky, P. (2002). Assessing student programming knowledge with web-based dynamic parameterized quizzes. In *ED-MEDIA 2002, Denver Colorado*.

Peter, F., Patel-Schneider, P. F., Hayes, P., Horrocks, I., & van Harmelen, F. (2003). Web ontology language (OWL) abstract syntax and semantics. W3C working draft. <http://www.w3.org/TR/2003/WD-owl-ref-20030331>.

Quevedo-Torrero, J. U., & Rodriguez-Bernal, A. D. (2007). Goal oriented templates to dynamically assess knowledge, information technology, 2007. In *ITNG'07. Fourth international conference on volume, issue, 2–4 April 2007* (pp. 953–954).

Rosić, M. (2000). *Establishing of distance education systems within the information infrastructure*. M.Sc. thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia (in Croatian).

Rosić, M., Glavinić, V., & Žitko, B. (2004). Intelligent authoring shell based on web services. In S. Nedevschi, & I. J. Rudas (Eds.), *Proceedings of the 8th international conference on intelligent engineering systems (INES 2004), Technical University of Cluj-Napoca, Cluj-Napoca, Romania* (pp. 50–55).

Santos, A. M., Simones dos Santos, P. A., Dionisio, F. M., & Duarte, P. (2002). On-line assessment in ungraduated mathematics. In *Second international conference on the teaching of mathematics, Greece, July 2002*.

Stankov, S. (1997). *Isomorphic model of the system as the basis of teaching control principles in an intelligent tutoring system*. PhD Diss., Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split, Split, Croatia, Dr. thesis (in Croatian).

Stankov, S. (2003). *Principal investigating project TP-02/0177-01 web oriented intelligent hypermedial authoring shell*. Ministry of Science and Technology of the Republic of Croatia, 2003–2005.

Stankov, S., Rosić, M., & Glavinić, V. (2001). Using quizzes in an intelligent tutoring systems. In *International summer school of automation, CEEPUS CZ_103, Maribor, Slovenia, June 10–22* (pp. 87–91).

Stankov, S., Rosić, M., Žitko, B., & Grubišić, A. (2008). TEx-Sys model for building intelligent tutoring systems. *Computers and Education*. doi:10.1016/j.compedu.2007.10.002.

Stankov, S., Žitko, B., & Grubišić, A. (2005). Ontology as a foundation for knowledge evaluation in intelligent e-learning systems. In *International workshop on applications of semantic web technologies for e-learning (SW-EL'05) in conjunction with 12th international conference on artificial intelligence in education (AI-ED 2005), July 18th, 2005, Amsterdam, The Netherlands* (pp. 81–84).

Wiener, N. (1948). *Cybernetics or control and communication in the animal and the machine*. New York: John Wiley.

Žitko, B. (2005). *Knowledge evaluation model in intelligent e-learning systems*. M.Sc. thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia (in Croatian).