# Using Schema Analysis for Feedback in Authoring Tools for Learning Environments

Harrie PASSIER & Johan JEURING
*Faculty of Computer Science, Open University of the Netherlands*
*Valkenburgerweg 177, 6419 AT Heerlen, The Netherlands*
*Email: harrie.passier@ou.nl & johan.jeuring@ou.nl*

**Abstract**. Course material for electronic learning environments is often structured using ontology and schema languages. During the specification and development of course material, many mistakes and errors can be made. In this paper we introduce schema-analysis as a technique to analyse structured documents, and to point out (possible) mistakes introduced by an author during authoring. With this technique we are able to produce valuable feedback. We show the technique at work using six categories of mistakes and two types of schemata.

## Introduction

Electronic learning environments (LE's), comprising eLearning systems, intelligent tutoring systems etc., are often complex tools. Since instances of such systems, for example for a particular course, are often written by non computer science experts, authoring tools have been developed to support the development of such courses. More open-ended authoring tools for LE's allow for more flexibility in both the form of the content and the order of the steps to design an LE [11]. This flexibility implies a higher probability of mistakes such as inconsistencies and inaccuracies. To improve the quality of LE's, an authoring tool should include mechanisms for checking the authored information on for example accuracy and consistency. Murray [11] mentions several such mechanisms. In this article we introduce schema-analysis, with which we are able to detect a number of the possible mistakes that can be introduced by an author during authoring a course.

The results presented in this paper are part of a project in which we will investigate general feedback mechanisms to learners as well as to authors [12]. To be able to produce semantically rich feedback we imagine an environment that contains several types of knowledge, like domain, task, education and feedback knowledge, which are represented by ontologies. These ontologies are the arguments of a general feedback engine, which observes student and author activities and matches these against the argument ontologies. This framework, a general feedback engine and the use of ontologies as arguments, supports the constant requirement for flexibility, adaptability and reusability of knowledge structures in LE's [2]. In this article we focus on feedback to support authoring LE's.

During authoring different aspects of a course, for example content, structure, and the ontologies, will be authored. In this article we focus on course structure and domain ontology, which we represent by IMS Learning Design (IMS LD) [8] and RDF.

Using IMS LD an author defines the structure of a course in a flexible way. IMS LD supports a wide range of pedagogies in online learning. Rather than attempting to capture the specifics of many different pedagogies, it does this by providing a generic and flexible language. With such a flexible language, an author can easily make mistakes. These



mistakes can be partly prevented by using templates. Some drawbacks of templates are: loss of flexibility, because an author must follow the steps prescribed by a template, and problems with maintainability: it is hard to maintain documents produced by means of templates. With schema-analysis we maintain flexibility, are able to produce feedback when an author makes a mistake, and leave the author, as a didactic professional, free to accept or not accept the feedback information [3]. The freedom to accept or not accept feedback is important. When a (possible) mistake is signalled, it is the author's decision to reject or accept the warning. Sometimes, it could be the author's intention to deviate from rules. What the system signals as a possible mistake may be correct from the author's point of view.

To determine the quality of a course, we want to detect whether or not the following properties hold for a course. If such a property holds, this may signal (the absence of) a potential mistake: (1) Completeness: Are all concepts that are used in the course defined somewhere? (2) Timely: Are all concepts used in a course defined on time? (3) Recursive concepts: Are there concepts defined in terms of it self? (4) Correctness: Does the definition of a concept used in the course correspond to the definition of the concept in the ontology? (5) Synonyms: Are there concepts with different names but exactly the same definition? (6) Homonyms: Are there concepts with multiple, different definitions?

Since a course and course related material are represented by means of schema languages such as IMS LD and RDF, we can use schema analysis techniques to answer the above questions, and to produce feedback about possible mistakes for authors. We have implemented the mentioned analyses as six distinct schema-analyses, which we show at work in a simple course structure and domain ontology. We have yet to develop examples in the context of real courses.

Schema analysis techniques are based, amongst others, on mathematical results about fixed points. Since these results are not widely known, we will explicitly show how to use them in the context of schema analyses. Schema analyses will be expressed in the functional, declarative, programming language Haskell, since this allows us to stay close to the mathematical results we use. We briefly explain Haskell, for more information see [7].

This article is structured as follows: Section 1 briefly explains what we mean with schemata and introduces the languages we use to represent them. Furthermore, we extend IMS LD to be able to define more structure. Section 2 introduces schema analysis. The Haskell constructs we use are introduced in section 2.1. Sections 2.2 and 2.3 presents the necessary data structures and algorithms. The last two sections discuss related work (section 3) and conclude (section 4).

## 1. Schemata and representations

An ontology specifies the objects in a domain of interest together with their characteristics in terms of attributes, roles and relations. Using an ontology many aspects of a certain domain can be represented, such as categories (taxonomic hierarchy), time, events and composition [13]. A composite object contains objects related to other objects using ' has_part' or 'uses' relations. Any object that consists of parts is called a composite object. A composite object has structure: the parts and their relations. Such a structure description is called a script or a schema. In this article we focus on schemata.

**Domain ontology -** To represent a domain ontology we use RDF, which can be used to represent meta-data as well as the semantics of information in a machine accessible way. RDF is a universal language that describes resources. The basic building block of RDF is a triple: <resource, property, value>, which defined concepts and related concepts. For example the concept 'cycle_wheel' consists of (has parts) the concepts 'rim' and 'spoke' (<cycle_wheel, has_part, rim> and <cycle_wheel, has_part, spoke>).

**Course structure** - XML is a language for structuring documents. A data type definition (DTD) describes the type of a set of XML documents. IMS LD [8] is a DTD developed to represent structures of e-courses. The content of a course is presented in a structured way, and activities in an activity-structure. For the examples in this paper we focus on the Activity-model, which consists of several elements: Metadata, Objectives, Prerequisites, Environment and an Activity-description. An activity-description consists of nine elements. One of them is the What-element, which contains the instruction for the activity to be performed. Possible instructions are grouped together by the parameter entity Extra-p. To be able to add more specific annotations to content and structure we introduce two new elements in the Extra-p element: *Definition* and *Example*. Furthermore, we introduce a new attribute *Educational-strategy* of the element Activity with two possible values: *Inductive* and *Deductive*. Introducing such elements will make it possible to structurally analyse educational material. These elements serve as examples to illustrate the analysis techniques at work. In practice many elements can be added, depending on the desired analyses. Listing 1 shows only the relevant elements and attributes related to the activity-model together with the newly defined elements example and definition. The new elements and attribute are marked in bold.

```
<!ELEMENT Activity  %Activity-model; >
<!ATTLIST Activity
        …
        Educational-strategy (Inductive | Deductive) >
<!ENTITY %Activity-model "(Metadata?, …, Activity-description)" >
<!ELEMENT Activity-description  (Introduction?, What, How?, …, Feedback-description?) >
<!ELEMENT What  %Extra-p; >
<!ENTITY %Extra-p "(…| Figure | Audio | Emphasis | List | … | Example | Definition)*" >
```

**Listing 1.** Parts of the activity-model in IMS LD definition

The definitions of the new elements *Definition* and *Example* are presented in listing 2.

```
<!ELEMENT Definition   (Description, Concept, RelatedConcept+) >
<!ATTLIST Definition Id ID    #REQUIRED
                    Name CDATA #REQUIRED>
<!ELEMENT Example (Description, Concept, RelatedConcept+) >
<!ATTLIST Example Id  ID    #REQUIRED
                    Name CDATA #REQUIRED
                    Belongs-to-definition IDREFS #REQUIRED>
<!ELEMENT Description (CDATA)>
<!ELEMENT Concept EMPTY>
<!ATTLIST Concept Id ID    #REQUIRED
                  Name CDATA #REQUIRED
<!ELEMENT RelatedConcept EMPTY>
<!ATTLIST Concept Id ID    #REQUIRED
                  Name CDATA #REQUIRED
```

**Listing 2.** Definition of the new elements

## 2. Schema analysis to detect authoring problems

The schemata given in Section 1 represent structural aspects, which can be analysed. In this section we give some examples of schema-analyses that determine whether or not certain properties hold. The results of these analyses form the basis of feedback to the author. The analyses take the schemata as input. In this paper we perform two types of analyses: 1) the analysis of structural properties of a schema, for example the recursive property, and 2) the



comparison of a schema with one or more other schemata, for example to test the correctness of a definition.

*2.1 Haskell preliminaries*

The *tuple* data type $(t_1, t_2, \ldots t_n)$ is constructed from component types. It consists of values $(v_1, v_2, \ldots, v_n)$, in which $v_1 :: t_1$, etc (where :: means 'is of type'). Function *fst* selects the first element of a pair, *fst* $(x, y) = x$, and function *snd* selects the second element. We use the data type *list* extensively. The empty list is denoted by [ ], and the concatenation of two lists $x$ and $y$ is denoted by $x ++ y$. Prepending an element $x$ to a list $xs$ is denoted by $x{:}xs$. In a list comprehension $[x \mid x \leftarrow xs, \textit{test}\ x]$, a new list is generated from the list $xs$. Each element $x$ of $xs$ is tested, and, if the test succeeds, added to the new list. Function *map f* takes a list and applies function $f$ to all elements in the list, so *map f xs* $= [f\ x \mid x \leftarrow xs]$. Anonymous functions can be constructed using lambda notation (\), so function $\backslash(x, y, z) \rightarrow (x, y)$ selects the first two components of a triple. Function *null* tests if a list is empty: *null* [ ] = *True*. To check if an element $x$ is an element of list $xs$, we use the expression *elem x xs*. Function *zip* takes two lists and returns a list of corresponding pairs: *zip* [1,2] [3,4,5] = [(1,3), (2,4)], where extra elements in the longer list are discarded. Functions *head* and *tail* extract the first and the remaining elements of a nonempty list, respectively. Function *inits* returns the list of initial segments of its argument list: *inits "abc"* == [*"","a","ab","abc"*], and function *tails* returns the list of all tail segments of its argument list: *tails "abc"* == [*"abc", "bc", "c",""*].
*Function composition* composes two functions: the output of the second function ($g$) becomes the input of the first function ($f$): $(f \cdot g)\ x = f\ (g\ x)$. The type of a function $f :: t_1 \rightarrow t_2 \rightarrow t_3$ can be read as: function $f$ takes two arguments of types $t_1$ and $t_2$ and returns a value of type $t_3$. Not all arguments have to be mentioned in a function definition, so *completeCourse c* = *null* (*undefinedConcepts c*), with type *completeCourse* :: *Course* → *Bool*, equals *completeCourse* = *null* . *undefinedConcepts*. Functions can be passed as parameters. For example, in *map isEven* [1,2,3,4] the type of *map* is *map* :: (*Int* → *Bool*) → [*Int*] → [*Bool*]. Choice between conditions is represented by a vertical bar '|'. For example

```
max x y | x >=y       = x
        | otherwise  = y
```

means: if the guard *x >=y* is true then return $x$, otherwise return $y$.

*2.2 Data structures and definitions*

We represent an ontology with a list of concepts. A concept is a tuple consisting of a concept identifier and a bag (multiset) of related concepts. In this definition we abstract from concept names, attributes and cardinalities.

```
data Ontology          = Ont [Concept]
type Concept           = (Id, RelatedConcepts)
type RelatedConcepts   = Bag
type Bag               = [Id]
type Id                = String
```

Note that a *data* type definition in Haskell introduces a constructor function for the data (*Ont* in the case of *Ontology*), whereas *type* definitions use the constructors of the types used. The structure of the data type Course follows the IMS LD definition (see listing 1) and consists of an identifier and a list of activities. Extra-p is limited to example and definition.

```
data Course            = C (Id, [Activity])
type Activity          = (Id, Estrategy, [Extra_p])
data Estrategy         = Inductive | Deductive
data Extra_p           = Ex  (Id, ConceptId, RelatedConcepts, DefRefs)
```

```
                        | Def (Id, ConceptId, RelatedConcepts)
type RelatedConcepts    =  Bag
type DefRefs            =  Bag
type ConceptId          =  Id
```

*terminalConcepts* are the set of concepts with no related concepts, *nonTerminalConcepts* the set of concepts with at least one related concept, and *allConcepts* the set of all concepts. Function *reachable* :: [*Concept*] → [*Concept*] → [*Concept*] takes as input: (1) a set of 'productions' (a set of concepts) and (2) a set of concepts, and returns for these last concepts the concepts that are reachable using the productions. Function *reachableTerminals* :: [*Concept*] → [*Concept*] → [*Concept*])) returns the set of terminal concepts that are reachable from the set of concepts using the productions.

**Example** – If *o* = *Ont* [(*a*, [*b*,*c*]), (*b*, []), (*c*, [*d*,*e*]), (*d*, []), (*e*, [])] :: *Ontology,* where the letters represent concepts, then:
*terminalConcepts* = [(*b*, []), (*d*, []), (*e*, [])], *nonTerminalConcepts* = [(*a*, [*b*,*c*]), (*c*, [*d*,*e*])], *allConcepts* = [(*a*, [*b*,*c*]), (*b*, []), (*c*, [*d*,*e*]), (*d*, []), (*e*, [])], *reachable nonTerminalConcepts allConcepts* = [(*a*, [*b*,*c*,*d*,*e*]), (*b*, []), (*c*, [*d*, *e*]), (*d*, []), (*e*, [])] and *reachableTerminals nonTerminalConcepts nonTerminalConcepts*= [(*a*, [*b*,*d*,*e*]), (*c*, [*d*,*e*])].

Functions *reachable* and *reachableTerminals* calculate a fixpoint by means of function *limitBy*. Function *reachable* is defined as:

```
reachable productions concepts =  limitBy equalConcepts  (expand productions)  concepts
```

Function *expand* expands *concepts* using *productions*: if production ($\beta$, $\omega$) is used, all related concepts $xs{+}{+}\beta\ {+}{+}ys$ are expanded to $xs{+}{+}\beta\ {+}{+}\omega{+}{+}ys$ removing duplicates. Function *limitBy* repeatedly applies function *expand* until a fixpoint is reached, in which each concept is bound to all concepts it can reach. Function *equalConcepts* checks whether or not two sets of bindings of reachable concepts are equal. A fixpoint is reached when *equalConcepts* returns true.

**Example** – Suppose *productions* = [(*a*, [*b*,*c*]), (*c*, [*d*,*e*])], *concepts* = [(*a*, [*b*,*c*]), (*c*, [*d*,*e*])] and function call *reachable productions concepts*. At the start *concepts* equals [(*a*, [*b*,*c*]), (*c*, [*d*,*e*])]. After one iteration it equals [(*a*, [*b*,*c*,*d*,*e*]), (*c*, [*d*,*e*])]. After the second iteration the value is unchanged: [(*a*, [*b*,*c*,*d*,*e*]), (*c*, [*d*,*e*])], so a fixpoint is reached and each concept is bound to the set of all reachable concepts from that concept.

Function *limitBy*, defined by

```
limitBy :: (a → a → Bool) → (a → a) → a → a
limitBy eq h s
  | eq s next  =  s
  | otherwise  =  limitBy eq h next
  where next = h s
```

only terminates if its argument function of type $a \to a$ is continuous on a *complete partial order* or *CPO* [6], which informally means that the argument function should be increasing on a restricted domain.

Function *reachableTerminals* is implemented in similar way. Instead of function *expand* function *derivationStep* is used as argument of function *limitBy*. Using production ($\beta$, $\omega$), all related concepts $xs{+}{+}\beta\ {+}{+}ys$ are replaced by $xs{+}{+}\omega{+}{+}ys$. More details about efficient algorithms for computing fixpoints for grammar analyses can be found in [9].

### *2.3 Solving authoring problems with schema analysis*

In this section we describe six algorithms (four briefly and two in more detail), which can be used to signal the (possible) mistakes listed in the introduction of this paper. The complete code can be obtained from:

http://www.ou.nl/info-alg-inf/Medewerkers/en_Passier.htm.



**Completeness** – We distinguish three kinds of (in)completeness: (1) within a course, (2) within an domain ontology and (3) between a course and an domain ontology. If a concept is used in a course, for example in a definition or an example, it has to be defined elsewhere in the course. The undefined concepts in a course are calculated in three steps: (1) determine the set of concept id's that appear in the right- and left hand sides of concepts within examples and all concept id's that appear in the right hand side of concepts within definitions (used concepts), (2) determine the concept id's that appear in the left-hand side of concepts in definitions (defined concepts) and (3) check that each of the used concepts appears in the set of defined concepts. A course is complete if all concepts used appear in the set of defined concepts. Completeness can also be applied to an (domain) ontology, and between a course and an ontology. The first one checks whether all used concepts in the ontology are defined in the same ontology, the second one if all used concepts in a course are defined in the ontology. The same three steps are performed in both functions.

**Timely** – A concept can be used before it is defined. This might not be an error if the author uses an inductive instead of a deductive strategy to teaching, but issuing a warning is probably helpful. Furthermore, there may be a large distance (measured for example in number of pages, characters or concepts) between the definition and the use of the concept, which is probably an error. We define the function *timely* to determine whether or not concepts in a course are defined in time and a function *outOfOrderConcepts* to list the concepts that appear to be out of order.

```
timely  ::  Course → Bool
timely  =  null . outOfOrderConcepts
```

In function *outOfOrderConcepts,* function *extractActivities* returns for every activity in the course the tuple (*Estrategy,* [*Extra_p*]) and puts these tuples in a list *activities*. Then, using functions *inits* and *tails* every [*Extra_p*] list is split as follows: for every element *x* in the list [*Extra_p*] the list is subdivided into a left part (*epl*), which contains all elements to the left of element *x,* and a right part (*epr*), which contains element *x* as and all elements to the right of *x*. For example, for the input list [*e*,*d*] we get [([], [*e*,*d*]), ([*e*], [*d*]), ([*e*,*d*], [])], where *e* is example and *d* is definition. Finally, function *intime* tests the timely constrains for all tuples (*es*, (*epl,epr*)): if the first element of *epr* is a definition and the educational strategy is deductive, then: 1) a related example appears after the definition, and 2) no related example appears before the definition (tested by *elemBy eqConcept c* in the code below). In case of an inductive activity, a related example appears before the definition and no related example appears after the definition. Function *intime* is always true if *epr* is empty or the first element of *epr* is an example.

```
outOfOrderConcepts    ::  Course → [Extra_p]
outOfOrderConcepts c  =
  let activities   = extractActivities c
      split        = [ (es,s) | (es,eps) <- activities, s <- zip (inits eps) (tails eps)]
  in [head epr | (es,(epl ,epr)) ← split, not (intime (es, epl, epr))]

intime (_,_,[])                              = True
intime (_,_,Ex (j, c, cs ,r):_)              = True
intime (Deductive, epl, Def (j, c  ,cs):epr) = elemBy eqConcept c epr && not (elemBy eqConcept c epl)
intime (Inductive, epl, Def (j ,c, cs):epr)  = elemBy eqConcept c epl && not (elemBy eqConcept c epr)

eqConcept id (Def (i,c,cs))     = False
eqConcept id (Ex (i,c,cs,r))    = id == c
```

**Recursive concepts** – A concept can be defined in terms of itself. Recursive concepts are often not desirable. If a concept is recursive, there should be a base case that is not recursive. Recursive concepts may occur in a course as well as in an ontology. We define two functions: *recursiveOntology* and *recursiveCourse* which take an ontology respectively a course as argument. Both first extract all concept definitions, and use function *recursiveConcepts*. We show the definition of *recursiveOntology*.

```
recursiveOntology :: Ontology → Bool
recursiveOntology = not . null . listRecursiveConceptsOntology

listRecursiveConceptsOntology :: Ontology → [Id]
listRecursiveConceptsOntology = recursiveConcepts . extractAllConceptsOnt
```

Function *recursiveConcepts* calculates for every concept all reachable concepts, as explained in section 2.2. Every concept in *reachables* is checked for recursiveness: a concept is recursive if the concept's Id is a member of the set of reachable concepts. The recursive concepts are collected in a list.

```
recursiveConcepts :: [(Id, RelatedConcepts)] → [Id]
recursiveConcepts allConcepts =
    let nonTerminalConcepts = filter (not . null . snd) allConcepts
        reachables          = reachable nonTerminalConcepts allConcepts
    in [x |(x, y) ← reachables, elem x y]
```

**Synonyms** – Concepts with different names may have exactly the same definition. For example, concept *a*, with concept definition (*a*, [*c,d*]), and concept *b*, with concept definition (*b*, [*c,d*]), are synonyms. In general, given a set *productions,* two concepts *x* and *y* are synomyms if their identifiers are different, $Id_x \neq Id_y$, and (*reachableTerminals productions x*) equals (*reachableTerminals productions y*).

We define function *synonyms* to check for synonyms in an ontology: for all concepts in the ontology all reachable terminal concepts are determined. Concepts with the same reachable terminal concepts and different concept id's are collected in a list.

**Homonyms** – A concept may have multiple, different definitions. If for example concept *a* has definitions (*a*, [*b,c*]) and (*a*, [*d,f*]), then these two definitions are homonyms. To list the homonyms in an ontology, we calculate the concepts that appear at least twice in the left hand side of a definition.

**Correctness** – The concepts in a course should correspond to the same concepts in its domain ontology. To determine whether or not this is the case, for every concept in a course all reachable terminal concepts are determined by function *reachableTerminals.* The set of productions contains the course's concepts completed with the concepts of the ontology for concepts that are not defined in the course. The result of this calculation is compared against the reachable terminal concepts of the same concept defined in the ontology.

## 3. Related work

Although many authors underline the necessity of feedback in authoring systems [1][3][11], we have found little literature about feedback and feedback generation in authoring systems. Jin et al [10] describe an authoring system that uses a domain as well as a task ontology to produce feedback to an author. The ontologies are enriched with axioms, and on the basis of the axioms the models developed can be verified and messages of various kinds can be generated when authors violate certain specified constraints. The details of the techniques used are not given, and it is not clear to us how general the techniques are. Our contribution is the introduction of schema analysis as a general technique to produce messages about errors of structural aspects of course material.

Aroyo et al. [1][2][3] describe a common ontology (web) based authoring framework. The framework contains a domain as well as a task ontology and supports an authoring process in terms of goals, and primitive and composite tasks. Based on ontologies, the framework monitors and assesses the authoring process, and prevents and solves inconsistencies and conflicting situations. Their requirements for authoring support are: (1) help in consistently building courseware, (2) discovery of inconsistencies and conflicting situations, (3) modularisation of authoring systems (reusability), (4) production of feedback, hints and recommendations, and (5) allow to accept or reject the proposed



solutions. We think that our framework satisfies all these requirements. Schema analysis as a technique could be positioned in (1), (2) and (4).

Stojanovic et al present an approach for implementing eLearning scenarios using the semantic web technologies XML and RDF, and make use of ontology based descriptions of content, context and structure [14]. A high risk is observed that two authors express the same topic in different ways (homonyms). This problem is solved by integrating a domain lexicon in the ontology and defining mappings, expressed by the author itself, from terms of the domain vocabulary to their meaning defined by the ontology. In our approach these mappings are analysed automatically.

In the Authoring Adaptive Hypermedia community the importance of feedback mechanisms in authoring systems has been recognized [5]. Although we have found an impressive amount of authoring tools for adaptive hypermedia [4], we have not found descriptions of technologies used for providing feedback to authors. We expect our results will be useful for authoring adaptive hypermedia as well.

## 4. Conclusions and further research

This paper shows schema analysis techniques to analyse structural aspects of learning environment related material, and applies these techniques to several example problems.

We have planned further research on the application of these ideas to support students developing artefacts in a specific domain using modelling languages like the unified modelling language (UML). Besides a domain and a course ontology we then also use a task and feedback ontology to produce feedback. Secondly, we have planned to define feedback patterns based on ontologies for educational elements such as certain types of questions, examples, definitions, etc. using IMS LD as modelling language. Thirdly, we have planned to test these analysis techniques in a real environment.

## References


[1] Aroyo L., Dicheva, D., Authoring support in concept-based web information systems for educational applications, in Int. J. Cont. Engineering Education and Lifelong Learning, Vol. 14, No. 3, 2004.

[2] Aroyo L., Dicheva D., The new challenges for e-learning: The educational semantic web, Educational technology & Society, 7 (4), 59 – 69, 2004.

[3] Aroyo, L. Mizoguchi, R., Towards Evolutional authoring support systems, Journal of interactive learning research 15(4), 365-387, AACE, USA, 2004.

[4] Brusilovsky, P. Developing adaptive educational hypermedia systems: From design models to authoring tools. In: T. Murray, S. Blessing and S. Ainsworth (eds.): Authoring Tools for Advanced Technology Learning Environment. Dordrecht: Kluwer Academic Publishers, 377-409, 2003

[5] Cristea, A., Authoring of Adaptive Hypermedia: Adaptive Hypermedia and Learning Environments, book chapter in "Advances in Web-based Education: Personalized Learning Environments", Sherry Y. Chen and Dr. George D. Magoulas. IDEA Publishing group, 2004.

[6] Davey B., Priestly H, Introduction to lattices and order, 2e edition, Cambridge University Press, 2001.

[7] Haskell: www.haskell.org

[8] IMS LD: http://www.imsglobal.org/learningdesign/index.cfm.

[9] Jeuring J., and Swierstra D., Constructing functional programs for grammar analysis problems, In Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture, pages 259--269, 1995.

[10] Jin L., Chen W., Hayashi Y., Ikeda M., Mizoguchi R., An ontology-aware authoring tool, Artificial intelligence in Education, IOS Press, 1999

[11] Murray, T., Authoring intelligent tutoring systems: An analysis of the state of the art. International Journal of AI in education, 10, 98 - 129, 1999.

[12] Passier, H., Jeuring, J., 2004. Ontology based feedback generation in design-oriented e-learning systems, Proceedings of the IADIS International Conferencee-Society 2004, Avila, Spain.

[13] Russell, S, Norvig, P., Artificial intelligence, A modern approach, Prentice Hall Int. editions, 1995.

[14] Stojanovic, L. Staab, S., Studer R., eLearning based on the semantic web, in WebNet 2001 – World conference on the www and internet, Orlando, Florida, USA, 2001.