The MATHESIS Semantic Authoring Framework: Ontology-Driven Knowledge Engineering for ITS Authoring

Dimitrios Sklavakis and Ioannis Refanidis

University of Macedonia, Department of Applied Informatics,
Egnatia 156, P.O. Box 1591, 540 06 Thessaloniki, Greece
{dsklavakis,yrefanid}@uom.gr

Abstract. This paper describes the MATHESIS semantic authoring framework being developed within the MATHESIS project. The project aims at an intelligent authoring environment for reusable model-tracing tutors. The framework has three components: an intelligent web-based model-tracing algebra tutor, an ontology and a set of authoring tools. The tutor serves as a prototype for the development of the ontology. The purpose of the ontology is to provide a semantic and therefore inspectable and re-usable representation of the declarative and procedural *authoring knowledge* necessary for the development of any model-tracing tutor, as well as of the declarative and procedural knowledge of the specific tutor under development. The procedural knowledge is represented via the *process model* of the OWL-S web services description ontology. Based on such an ontological representation, a suite of authoring tools is being developed at the final stage of the project.

Keywords: authoring systems, ontologies, semantic web, intelligent tutoring systems, model-tracing, web based tutors, OWL-S.

1 Introduction

Intelligent tutoring systems (ITSs) and especially model-tracing tutors have been proven quite successful in the area of mathematics. Despite their efficiency, these tutors are expensive to build both in time and human resources [1]. The main reason for this is the classic *knowledge acquisition bottleneck*: the extraction of knowledge from domain experts, the representation of this knowledge and the implementation of it in effective ITSs.

As a solution, authoring programs have been built having as their main purpose the reduction of development time as well as the lowering of the expertise level required to build a tutor. An extensive overview of these authoring tools can be found in [2]. Still, these tools suffer from a number of problems such as isolation, fragmentation, lack of communication, interoperability and re-usability of the tutors that they build.

The main goal of the ongoing MATHESIS project is to develop authoring tools for model-tracing tutors in mathematics, with knowledge re-use as the primary characteristic for the authored tutors as well as for the authoring knowledge used by

A. König et al. (Eds.): KES 2011, Part II, LNAI 6882, pp. 114–123, 2011.

the tools. For this reason, in the first stage of the MATHESIS project the MATHESIS Algebra tutor was developed to be used as a prototype target tutor [3]. In the second stage, based on the knowledge used to develop the MATHESIS algebra tutor, an initial version of the MATHESIS ontology has been developed [4]. The MATHESIS ontology is an OWL ontology developed with the Protégé OWL ontology editor. As this initial version of the ontology was developed in a bottom-up direction, it emphasized on the representation of the tutor, namely the interface, tutoring and domain expertise models. Of course, the ontology contained also a representation of the authoring knowledge but in a rather conceptual level. The project is now in its third stage were the authoring tools are being developed. These include tools for developing: a) the declarative tutoring knowledge (interface, cognitive tasks), b) the procedural tutoring knowledge (domain expertise and tutoring models) and c) the authoring knowledge.

This paper describes these authoring tools, how they are integrated with the MATHESIS ontology and how they operate on it to build both a model of the tutor under development which can be parsed and produce the tutor's code (HTML and JavaScript) as well as an *executable* model of the authoring processes that produced the tutor's model. It must be stressed out that because of the complex intertwining of the MATHESIS ontology with the MATHESIS authoring tools, the development of the authoring tools led to the following modifications of the initial ontology: a) the Document Object Model (DOM) representation of the tutor's HTML interface was simplified, b) the representation of the procedural tutoring knowledge (domain and tutoring model) was simplified and refined by introducing a taxonomy of atomic processes representing the various JavaScript statements, c) the authoring knowledge was developed as a full *executable model authoring language*, OntoMath, with composite authoring processes and atomic authoring statements.

The rest of the paper is structured as follows: Section 2 describes the newer, refined and extended version of the MATHESIS ontology as it is integrated with the authoring tools. Section 3 describes the authoring tools developed on top of this ontological representation. Section 4 presents some related work and, finally, Section 5 concludes with a discussion and further work for the development of the MATHESIS framework.

2 Semantic Knowledge Representation in the MATHESIS Framework: The MATHESIS Ontology

The main component of the MATHESIS authoring framework is the MATHESIS Ontology. The ontology contains three kinds of knowledge: a) the declarative knowledge of the tutor, such as the interface structure and student models, b) the procedural knowledge of the tutor, such as the teaching and math domain expertise models and c) the authoring knowledge, i.e. the declarative and procedural knowledge that is needed to develop the other two kinds. While the declarative knowledge is represented with the basic OWL components, i.e. classes, individuals and properties, the procedural knowledge, both tutoring and authoring, is represented via the *process model* of the OWL-S web services description ontology. By using OWL-S, every authoring or tutoring task is represented as a *composite process*.

2.1 Procedural Knowledge Representation: The OWL-S Process Model

OWL-S is a web service description ontology. Every service is an instance of class Process. There are three subclasses of Process, namely the AtomicProcess, CompositeProcess and SimpleProcess (not actually used). Atomic processes correspond to the actions a service can perform by engaging it in a single interaction; composite processes correspond to actions that require multi-step protocols.

Composite processes are decomposable into other composite or atomic processes. Their decomposition is specified by using control constructs such as Sequence and If-Then-Else. Therefore, any composite process can be considered as a tree whose non-terminal nodes are labeled with control constructs, each of which has children specified using the components property. The leaves of the tree are invocations of other processes, composite or atomic. These invocations are indicated as instances of the Perform control construct. The process property of a Perform indicates the process to be performed. In Figures 3 and 4 (Section 3) the tree structures of three different composite processes can be seen, as they are displayed to the user by the MATHESIS authoring tools. This tree-like representation of composite processes is the key characteristic of the OWL-S process model used in the MATHESIS ontology to represent authoring, tutoring and domain *procedural* knowledge *declaratively*, as it will be described in the following sections.

2.2 Tutor Representation in the MATHESIS Ontology

The MATHESIS project has as its ultimate goal the development of authoring tools that will be able to guide the authoring of real-world, fully functional model-tracing math tutors. The MATHESIS Algebra tutor is a Web-based, model-tracing tutor that teaches expanding and factoring of algebraic expressions. It is implemented as a simple HTML page with JavaScript controlling the interface interaction with the user and implementing the tutoring, domain and student models. The user interface consists of common HTML objects as well as Design Science's WebEQ Input Control applet, a scriptable editor for displaying and editing mathematical expressions.

On the top level of the MATHESIS ontology, every tutor is represented as an instance of class ITS_Implemented, having two properties: a) hasDomainTask which keeps a list of Domain_Task instances, the math tasks that the tutor teaches, e.g. monomial_multiplication, and b) hasTopInterfaceElement which keeps the root of the HTML DOM, a Document instance. Every instance of Domain_Task, like monomial_multiplication, has three properties: a) hasTutoringModel which keeps an instance of ITS_Teaching_Model, the algorithm that the tutor uses to teach this task, e.g. Model_Tracing_Algorithm, and b) hasInputKnowledgeComponents and hasOutputKnowledge-Components which keep lists of Domain_Knowledge_Component instances, the math concepts given and asked for the task. For the monomial_multiplication task, the input knowledge components are two monomials and the output knowledge component is their product, a monomial too. Each Domain_Knowledge_Component like monomial, has two properties: a) hasInterfaceElement which keeps a list of HTMLObject instances, the HTML interface object(s) used to display the structure of the concept, e.g.

WebEQ_Input_Control and b) hasDataStructure which keeps a list of the programming data structures used to represent the concept in the tutor's JavaScript code.

The top level representation of the tutor's procedural knowledge in the ontology is the model-tracing algorithm represented as a composite process, named ModelTracing-Algorithm. The tree structure of the process, adapted for the monomial_multiplication task, is shown in Figure 3a (Element 9) as displayed by the authoring tools. Each step of the algorithm is a top level tutorial action: present the current problem solving state, get the correct solution(s) from the domain expertise model, provide help/hint, get the student solution, provide feedback, assess, discuss. Each of these steps is also a composite process analyzing the tutorial steps further down to more simple ones like giving a hint, showing an example, recalling math formulas or rules and so on. In programming terms, this composite process when translated to JavaScript code will be the main function that controls the whole tutoring process by calling other functions.

This recursive analysis ends when a composite process contains only atomic processes corresponding to JavaScript statements. For the monomial_multiplication task, the execute_monomial_multiplication-Execution process is analyzed in two other composite processes: multiplyCoefficients and multiplyMainParts. These two processes form the tutor's *domain expertise model*, which calculates the correct answer(s) in each step in order to be compared against the student's answer. Figure 1 shows the structure of process multiplyMainParts.

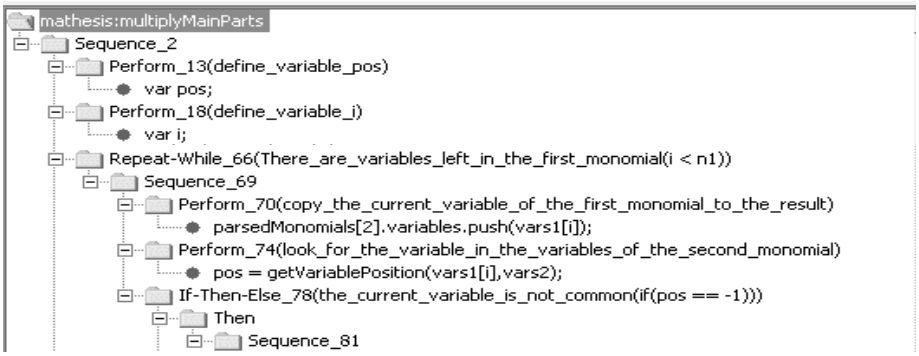

Fig. 1. Representation of the JavaScript function multiplyMainParts

Each JavaScript statement is represented by an instance of the JavaScriptStatement class, a subclass of AtomicProcess. Following the OWL-S representational scheme, these instances are parameters to Perform constructs. The JavaScriptStatement class has subclasses which classify the JavaScript statements in various classes such as DefineVariable, InitializeVariable, AssignValueToVariable, InvokeFunction, InvokeMethod, SetProperty. Each subclass has properties that represent the various parts of the corresponding JavaScript statements. For example, the InvokeFunction class has three

properties, hasInvokedFunction, hasArgumentsList and hasAssignedVariable. From the values of these properties the authoring tools create and display the actual JavaScript code as shown in Figure 1 (the dots under the Perform constructs). Such a detailed model of the JavaScript language allows the authoring processes to guide the non-expert author in building the tutor's code by selecting the appropriate JavaScriptStatement subclass and the values of the related properties. Therefore, a non-expert author does not need to know the JavaScript syntax but only have some general programming knowledge. As far as it concerns the semantic validation of the produced JavaScript code, the tools do not provide any special assistance to the authors. Therefore, the produced JavaScript code is syntactically correct but whether this code produces the intended behavior is a matter of correct design and analysis of the tutoring processes. In turn, this is a matter of correct design and analysis from the authors' part as is the case with any other system implementation.

At last, the representation of the HTML code and the corresponding Document Object Model (DOM) of the user interface are shown in Figure 2. Each object defined in the HTML code is represented as an instance of the corresponding HTMLObject subclass (Head, Body, Div, Input). Each instance has the corresponding HTML properties (body-onload, input-type). The DOM tree is represented via the two properties hasFirstChild and hasNextSibling. This representation allows for bi-directional creation of the HTML part of the user interface: a) The author creates in the ontology the representation of the DOM and then by traversing the DOM tree, the authoring tools generate the corresponding HTML code (top-down) or b) The user interface is created using any Web-page authoring program, the HTML file is parsed by Java's XML parser creating a DOM structure which in turn is transformed into its corresponding ontological representation for further authoring (bottom-up).

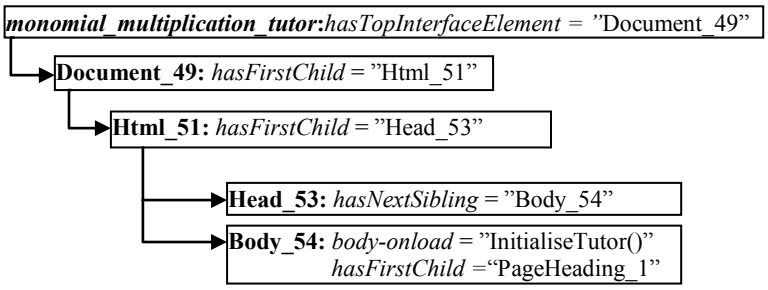

Fig. 2. The HTML User Interface Document Object Model Representation

2.3 Tutor Authoring Knowledge Representation

Within the MATHESIS framework, any authored tutor is represented in the MATHESIS ontology as described in the previous section. Expert authors have to create this ontological representation of the tutor. This is done by using the Protégé OWL interface to create classes, individuals, properties and values. The MATHESIS framework allows expert authors to capture the whole authoring effort by providing an *executable authoring model building* language, OntoMath. In OntoMath, each

authoring step is represented as an authoring process, composite or atomic. The authoring steps of a composite authoring process can be further decomposed in other authoring processes in exactly the same way that was described for tutoring processes in the previous section. Composite authoring processes correspond to functions of a programming language that can be called, getting and returning values. This is achieved by two properties: a) each composite process has a property, hasFormalParameters, that keeps a list of the process formal parameters, b) each Perform - the construct used to call a composite process – has a property, hasRealParameters, that keeps the list of the parameters at call time. During execution of a Perform construct, the interpreter matches the values of the real parameters to these of formal parameters. The values of the two properties are defined by the author in the ontology with the help of the authoring tools.

The recursive analysis of composite authoring processes ends to atomic authoring processes which are instances of the OntoMathStatement atomic process subclass. Each OntoMathStatement instance corresponds to an operation that must be performed to the MATHESIS Ontology such as create class, create instance, create property, get/set property value. OntoMath statements are *grounded* to actual Java program code. When the MATHESIS authoring tools interpret a Perform construct that calls an OntoMath statement, they execute its corresponding Java code, which performs various operations on the ontology that represents the tutor under development. It must be noted that these statements are not fixed. Expert authors can define their own atomic authoring statements by a) using the tools to define in the ontology the values of property hasFormalParameters for the new statement and b) writing the Java code that during execution gets the values of property hasRealParameters of the calling Perform construct and performs the statement's intended operation(s). The interpretation and execution of the OntoMath code by the MATHESIS authoring tools, as it will be described in section 3, leads to the creation of the ontological representation described in section 2.2, and therefore to the implementation of the authored tutor.

Therefore, the OntoMath authoring processes form a *meta-program* that handles the ontological representation of the tutor as its data. They capture the authoring expertise of expert authors and make it available to non-expert authors. They are the expertise model of an ITS authoring shell that, when executed, it guides non-expert authors to develop their own tutors.

3 The MATHESIS Authoring Tools

The MATHESIS authoring tools are currently implemented as a tab widget in Protégé. The tools are grouped in three windows according to their functionality: the Tutor authoring window, the Authoring processes authoring window (Author) and the MATHESIS ontology tab (Figure 3). The general philosophy of the tools is "point and click": the author selects a class or an instance in the ontology and clicks on a tool to perfrom an authoring action with it.

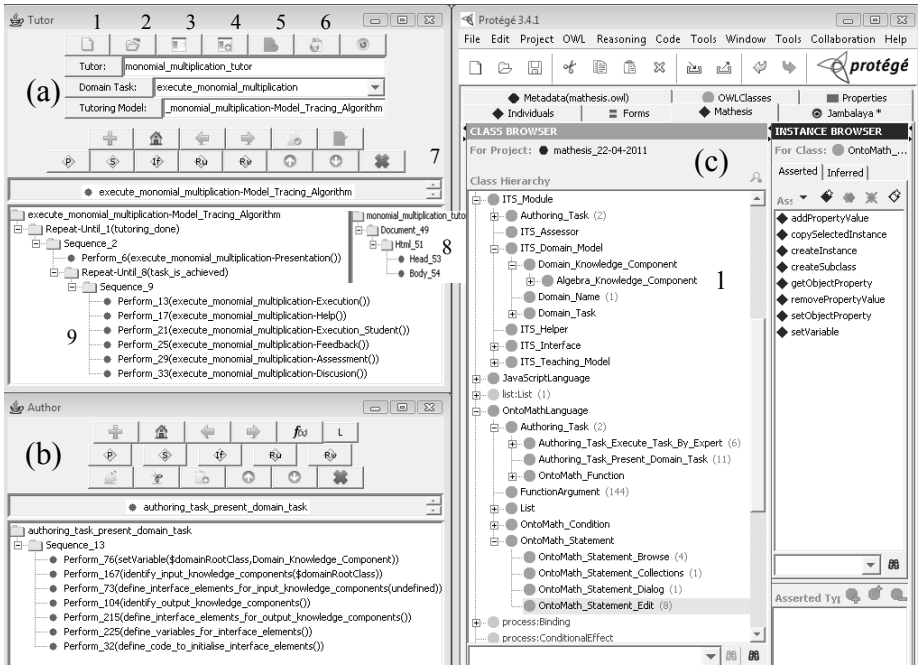

Fig. 3. The MATHESIS Authoring Tools: (a) Tutor window, (b) Authoring Processes Window, (c) The MATHESIS Ontology Tab

The Tutor authoring tools lie in the Tutor window (Figure 3a). With these tools an author can:

- Create a new instance of class ITS_Implemented for a new tutor (button a1) or select an existing one (button a2). In Figure 3a, the monomial_multiplication_tutor has been selected.
- Create a new or select an existing domain task that the tutor will teach (buttons a3, a4). New domain tasks are added as values to the hasDomainTask property of the tutor. The assigned tasks are displayed in a drop-down list. In Figure 3a the execute_monomial_multiplication task has been chosen to be authored.
- Assign to the selected domain task a generic tutoring model (button a5). Currently, only the Model_Tracing_Algorithm is defined in the ontology. The authoring tools create a new tutoring process by copying the structure of the generic tutoring process. In Figure 3a, the execute_monomial_multiplication-Model_Tracing_Algorithm has been assigned. Its structure is displayed by the tools (a9). As explained in section 2.2, the tutoring model is the main tutoring process from which all other processes are called. It is from this process that the authoring of the tutor starts.
- Switch views between the tutoring model (a9) and DOM representation of the tutor's HTML interface (a8). In reality the two structures are displayed separately.

The Tutor window also contains the *Tutoring Processes Authoring Tools,* arranged in two rows (a7). The tools of the first row allow the author to browse through the

tutoring processes that represent in the ontology the tutor's program code: when a Perform construct is selected, the author can move to the Performed (called) composite process. In the same time, a call stack is maintained so that the author can return back to the calling process. These tools are to be used by expert and non-expert users alike.

The second row of the *Tutoring Processes Authoring Tools* contains advanced tools, used mainly by expert authors. These tools allow the author to create and edit directly the tutor's processes (code). The reason for this is that some tutoring processes demand a very high level of authoring skills to be created. Expert authors can create these processes and provide them to non-expert authors as libraries. Examples of such tutoring processes are the model-tracing algorithm or an algorithm for parsing the MathML presentation code of an algebraic expression from the WebEQ Input Control applet.

Non-expert authors create tutoring processes by executing the corresponding authoring processes. This binding is done by the expert authors who set the hasAuthoringProcess property of a tutoring process through the advanced tools. When the hasAuthoringProcess property of a tutoring process points to an authoring process, with a click of a button the authoring process is displayed in the Author window. In Figure 3a, the tutoring process execute_monomial_multiplication-Presentation() has as a corresponding authoring process the authoring_task_present_domain_task, displayed in the Author window (Figure 3b). From this point, the next step is to execute this authoring process that will implement the tutoring process execute-monomial-multiplication-Presentation().

The Authoring Processes Tools lie in the Author window, shown in Figure 3b. They provide the same creating and editing operations for the authoring processes that form the *executable authoring model* of the implemented tutor. Their significant difference lies in the tracing (execution) of the Authoring Processes. When a composite authoring process is Performed (called) its tree structure appears in the Author window and its code is executed. When an atomic authoring process is Performed its corresponding Java code is executed.

As an example, the execution of the identify_input_knowledge_components authoring process (Fig. 4) will be traced. This authoring process is called by authoring process authoring_task_present_domain_task (Fig. 3b) and guides the non-expert author to select the domain concepts considered to be known for the currently selected domain task, execute_monomial_multiplication, of the authored tutor, monomial_multiplication tutor (Fig. 3a). For the monomial multiplication, the given domain concepts are two instances of class monomial. The OntoMath code is executed as follows:

1. Statement Perform_45(setSelectedClass($domainRootClass)) is performed. The interpreter gets the value of $domainRootClass, which is Domain_Knowledge_Component, and sets this class as selected in the Class Browser of the MATHESIS ontology tab (Fig. 3c). The author must browse into the subclasses of this class to find the monomial instance. Let's suppose that the instance exists (created by an expert author) and it has been found by the author.
2. Construct Repeat-While_2 starts an iteration executed for each knowledge component the author wants to identify.
3. Construct If-Then-Else_1 is executed having as its condition the authorConfirmation OntoMath predicate. To evaluate this predicate the interpreter displays a Yes/No question to the author ("Does the knowledge component already exist?").

According to step 1, the answer is "Yes", the predicate is true and execution continues with the "Then" part, i.e. Sequence_20.

4. Statement Perform_47(showMessageDialog(…)) prompts the author to select the existing domain concept in the Instance Browser of the MATHESIS ontology tab.
5. In construct If-Then-Else_22 a new instance of monomial is created, marked as selected and stored in variable $selectedInstance.
6. Statements Perform_55(getObjectProperty(…)) and Perform_58(setObjectProperty(…)) add the newly created instance of monomial to the list of values of property hasInputKnowledgeComponents of the execute-monomial-multiplication domain task.
7. Steps 3-7 are repeated until loop Repeat-While_2 is interrupted by the author, and the execution of the identify-input-knowledge-components authoring process is completed.

Fig. 4. The identify_input_knowledge_components authoring process

4 Related Work

The use of ontologies and semantic web services in the field of ITSs is relatively new. Ontological engineering is used to represent learning content, organize learning repositories, enable sharable learning objects and learner models, facilitate the reuse of content and tools [5]. The most relevant work to the MATHESIS framework is the OMNIBUS/SMARTIES project [6]. The OMNIBUS ontology is a heavy-weight ontology of learning, instructional and instructional design theories. Based on the OMNIBUS ontology, SMARTIES (SMART Instructional Engineering System) is a theory-aware system that provides a modeling environment and guidelines for authoring learning/instructional scenarios.

While the OMNIBUS/SMARTIES system provides support mainly for the design phase of ITS building, the MATHESIS framework aims at the analysis and development phases. It provides a semantic description of both tutoring and authoring

knowledge of any kind in the form of composite processes and the way to combine them as building blocks of intelligent tutoring systems. Thus, it provides the ground for achieving reusability, shareability and interoperability.

5 Discussion and Further Work

It is well known that the development of ITSs demands a considerable effort. The ontological representation of both the developed tutor and the authoring knowledge to develop it, definitely puts extra effort to the authoring endeavor. We believe that this extra effort pays off for the following reasons: a) the semantic representation of the tutor makes all parts of it open to inspection by other authors and therefore reusable, shareable and interoperable, b) the same holds for the authoring knowledge used to build the tutor and c) the OntoMath language is a proper programming language, completely open and configurable, therefore authoring processes can in principle build open, shareable and reusable authoring models for any kind of tutor.

Before this extra effort starts paying off, the MATHESIS semantic authoring framework must represent in its ontology a considerable amount of authoring and tutoring knowledge that now lies hidden and fragmented inside the various authoring tools and the authored tutors. As a first step, this will be done by representing a considerable part of the MATHESIS Algebra tutor into the ontology. For this purpose new authoring tools are needed: parsers for transforming between HTML and MATHESIS DOM representation; parsers for transforming between JavaScript and MATHESIS tutoring processes; extension of the OntoMath language and elaboration of its interpreter. These tools constitute our current research line.

References

1. Koedinger, K.R., Anderson, J.R., Hadley, W.H., Mark, M.A.: Intelligent Tutoring Goes to School in the Big City. International Journal of Artificial Intelligence in Education 8, 30–43 (1997)
2. Murray, T.: An overview of intelligent tutoring system authoring tools: Updated Analysis of the State of the Art. In: Murray, Ainsworth, Blessing (eds.) Authoring Tools for Advanced Technology Learning Environments, pp. 491–544. Kluwer Academic Publishers, Netherlands (2003)
3. Sklavakis, D., Refanidis, I.: An Individualized Web-Based Algebra Tutor Based on Dynamic Deep Model-Tracing. In: Darzentas, J., Vouros, G.A., Vosinakis, S., Arnellos, A. (eds.) SETN 2008. LNCS (LNAI), vol. 5138, pp. 389–394. Springer, Heidelberg (2008)
4. Sklavakis, D., Refanidis, I.: Ontology-Based Authoring of Intelligent Model-Tracing Math Tutors. In: Dicheva, D., Dochev, D. (eds.) AIMSA 2010. LNCS, vol. 6304, pp. 201–210. Springer, Heidelberg (2010)
5. Dicheva, D., Mizoguchi, R., Greer, J. (eds.): Semantic Web Technologies for e-learning, The Future of Learning, vol. 4. IOS Press, Amsterdam (2009)
6. Mizoguchi, R., Hayashi, Y., Bourdeau, J.: Inside a Theory-Aware Authoring System. In: Dicheva, D., Mizoguchi, R., Greer, J. (eds.) Semantic Web Technologies for e-learning, The Future of Learning, vol. 4, pp. 59–76. IOS Press, Amsterdam (2009)