The MATHESIS Meta-Knowledge Engineering Framework: Ontology-driven Development of Reusable Knowledge Engineering and Domain Expertise

Dimitrios Sklavakis* and Ioannis Refanidis

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

Abstract: The effect of the knowledge acquisition bottleneck is still limiting the widespread use of knowledge-based systems (KBS), especially in the area of model-tracing tutors, as they demand the development of deep domain expertise, tutoring and student models. The MATHESIS meta-knowledge engineering framework for model-tracing tutors, presented in this article, aims at maximizing knowledge reuse. This is achieved through ontological representation of both the declarative and procedural knowledge of a KBS (model-tracing tutor), as well as of the declarative and procedural authoring knowledge of the process to develop a KBS. Declarative knowledge is represented in Ontology Web Language (OWL). Procedural knowledge is represented using the concepts of atomic and composite processes of OWL-S web services description ontology. The framework provides knowledge engineering tools, integrated into the Protégé OWL ontology editor, for the development and management of the KBS's ontological representation. It also provides meta-knowledge engineering tools for the ontological representation of the knowledge engineering expertise as a set of composite knowledge engineering processes and atomic knowledge engineering statements. The latter constitute a language, ONTOMATH, for building executable knowledge engineering models that, when executed by the tools, guide non-expert knowledge engineers like domain experts to the creation of new knowledge-based systems (model-tracing tutors). The framework, being in an experimental stage, was used for the development of a monomial multiplication and division tutor. However, the overall design and implementation aimed at constituting the framework as a proof-of-concept system that can be used for the meta-knowledge engineering of more complex model-tracing tutors.

Keywords. Meta-knowledge engineering, knowledge engineering tools, intelligent tutoring systems, model-tracing tutors

1. Introduction

Intelligent tutoring systems (ITS), particularly model-tracing tutors (MTT), have been proven quite successful in the area of mathematics (Koedinger, Anderson, Hadley, & Mark, 1997; Koedinger & Corbett, 2006). Despite their effectiveness (Corbett 2001), these tutors are expensive to build both in time and human resources (Aleven, McLaren, Sewall, & Koedinger, 2006). Studies have shown that the development cost for one hour of teaching with a MTT is 200-300 hours (Koedinger, Anderson, Hadley, & Mark, 1997; Murray, 2003).This is due to the well-known *knowledge acquisition bottleneck* (Hoffman 1987), comprising the extraction of knowledge from domain experts, the representation of this knowledge and its implementation in effective knowledge-based systems (KBS).

Knowledge acquisition and, its counterpart, knowledge reuse have been proven to be the key problems for the development of expertise models, the models that represent and produce the problem-solving knowledge in knowledge-based systems. The main consequences are:

- High development demands in human resources, time and money.
- Demand for knowledge engineers possessing significant expertise.

* Corresponding author: Dimitrios Sklavakis, Department of Applied Informatics, University of Macedonia, Egnatia 156, P.O.Box 1591, 54006 Thessaloniki, GREECE. E-mail: dsklavakis@uom.gr

- Shallow, incomplete or even incorrect expertise models.
- Difficulties in modifying and/or expanding the expertise models.
- Inability to reuse developed expertise models in similar or new knowledge-based systems (an effect described as "re-inventing the wheel").

In the case of MTTs, the knowledge acquisition bottleneck gets even more serious as these systems must contain two expertise models:

1. The domain expertise model or problem solver, which represents the problem-solving knowledge of the tutored domain. This model is used to produce the valid solution steps of the tutored problem and allow the tutor to provide guidance and feedback to the student.
2. The pedagogical or tutoring model, which represents the teaching knowledge of the system such as how to present the problem, what problem-solving tools to provide to the students for entering their solution steps, when and how to give help, what kind of help/guidance to give, etc.

In turn, these models affect directly the design of the *user interface model*, which orchestrates the interactions between the aforementioned two models to produce the desired tutoring behaviour. In addition, some MTTs require the development of another model, the *student model*, which represents students' mastery of the tutored domain. This model is used by the system to provide student-adapted tutoring either within problems (micro-adaptation) or between problems (macro-adaptation).

The most difficult model to build is the domain expertise model. At the same time, it is the most critical one since it defines:

1. The tutor's *breadth*, that is, how many domain skills it can teach.
2. The tutor's *depth*, that is, how complex skills, in terms of the sub-skills contained, it can teach.
3. The tutor's *granularity*, that is, how fine-grained are the solution steps that the tutor can produce and guide.
4. The tutor's *scalability*, that is, the ability to reuse the tutor's domain expertise model for extending its breadth and depth.

We were confronted with the knowledge acquisition bottleneck when we decided to develop a model-tracing tutor to teach algebraic operations. In the Greek educational system, algebraic expressions and their operations are taught in the 3rd grade of Junior High School (ages 14-15). The curriculum covers the following mathematical operations: monomial multiplication, division and power; monomial-polynomial and polynomial-polynomial multiplication; parentheses elimination; collect like terms; identities (square of sum and difference, product of sum-difference, cube of sum and difference); factoring (common factor, identities, trinomial); combination of factoring methods; and operations of rational expressions.

Furthermore, we wanted that the domain expertise model of our tutor would be extensible so that it could cover new math sub-domains like second degree and rational equations solving. The solution of these kinds of equations usually demands the transformation of the original equations using the algebraic operations listed above. In our search for an authoring framework and tools that could support this endeavour, we realized that despite the efforts, advancements and successes in the currently developed authoring frameworks and the corresponding tutors, these frameworks have worked around the knowledge acquisition problem rather than confronting it directly. As a consequence, most of the developed tutors suffer from limited depth and breadth, whereas those having broader and deeper domain expertise models suffer from scalability issues (see Section 2). This was our motivation to deal directly with the knowledge acquisition problem in order to produce tutors that cover broader and more complex domains in a scalable way.

The rest of the article is structured as follows: First we present the background of our work consisting of an overview of the state-of-the-art in authoring (knowledge engineering) frameworks for Intelligent Tutoring Systems, the tutors produced and how they suffer from the knowledge acquisition bottleneck, coupled with a description of how the MATHESIS meta-knowledge

engineering framework provides the means to deal with this problem by using the results of research in the ontological engineering field. Then, we present the prototypical MATHESIS Algebra Tutor. Next, we present an overview of the MATHESIS meta-knowledge engineering framework followed by its key characteristic, the representation of procedural knowledge engineering expertise within the MATHESIS ontology as an executable knowledge engineering model. Then, we describe how expert and non-expert knowledge engineers (domain experts) use the meta-knowledge engineering and knowledge engineering tools to build the tutor's ontological representation. Finally, we discuss the results and identify potential directions of further work.

2. Background

In this Section we present an overview of the state-of-the-art in authoring (knowledge engineering) frameworks for Intelligent Tutoring Systems and the tutors produced, focusing on the knowledge acquisition bottleneck issue. Then we show how the MATHESIS meta-knowledge engineering framework provides the means to deal with this problem.

2.1. Related work

The most successful and widely used math model-tracing tutors are Cognitive Tutors developed by Carnegie Learning[1], based on more than twenty years of cognitive science research at CMU (Koedinger & Corbett, 2006). Cognitive Tutors are now an integral part of complete curricula used in hundreds of middle and high schools throughout the United States. Cognitive Tutors have to adapt to very strict guidelines and educational goals of the US educational system. Thus, they follow the textbook by teaching specific exercises that train the students in specific, simple domain tasks that don't contain other sub-tasks. Each problem has its own simple domain model and interface. Therefore, there is actually a set of independent tutors with narrow and shallow domain models, with different interfaces and not one tutor with a broad and deep domain model and a common interface. Concerning their scalability, for each problem the set of anticipated steps is *precomputed* by solving the problem in all acceptable ways by running a rule-based problem-solver (Van Lehn, 2006). Therefore, only the knowledge engineers can add new problems and for each one they must pre-program its solution steps. In contrast, the MATHESIS Algebra Tutor has a domain (math) expertise module that parses and solves each exercise producing the correct solution steps in real time (see Section 3). Carnegie Learning uses a proprietary knowledge engineering tool, the Cognitive Tutor SDK (Blessing, Gilbert, Ourada & Ritter, 2009), which supports the development of domain models based on the ACT Theory of cognition (Anderson, 1993). Problem solving states are represented by a hierarchy of *goalnode* instances with their *properties* and values, while problem solving steps are represented by a hierarchy of predicates that operate on the goalnodes. No information is given on how broad and deep these domain models can be or if they can be reused between the various tutors developed. The MATHESIS framework provides free, open authoring tools (Protégé) and representational schemes (OWL, OWL-S).

A publicly available set of authoring (knowledge engineering) tools for Cognitive Tutors are the Cognitive Tutors Authoring Tools (CTAT[2]) developed at the Human-Computer Interaction Institute of Carnegie Mellon University (Aleven, McLaren, Sewall, & Koedinger, 2006). After 8 years of use, CTAT is the most mature and widely used authoring tool. It supports two types of tutors, cognitive tutors, which were described above, and *example-tracing tutors* (Aleven, McLaren, Sewall, & Koedinger, 2009). While cognitive tutors have a cognitive model, implemented as a set of production rules in Jess[3], example-tracing tutors have a "generalized example" of the solution of a specific problem, implemented as a "behavior graph", an acyclic graph where nodes represent problem-solving states and links represent problem-solving steps. Example-tracing tutors are authored using a programming-by-demonstration technique by creating

[1] www.carnegielearning.com
[2] http://ctat.pact.cs.cmu.edu
[3] http://www.jessrules.com

initially a tutor interface for the targeted problem type through drag-and-drop techniques, then demonstrating through this interface the problem's solution and finally editing, annotating and generalizing the resulting behavior graph. In the case of cognitive tutors, the last step demands the development of the cognitive model implemented as production rules in Jess by AI programmers.

ASTUS[4] is a framework for domain independent model-tracing tutors development. It is designed to provide a knowledge representation language for the development of the domain model richer than that of CTAT (Paquette, Lebeau, & Mayers, 2010). The purpose is to model domains from a pedagogical perspective rather than a cognitive one, allowing experimentation with varied pedagogical strategies. The framework is relatively new and the knowledge engineering language is not yet fully developed, with only a few tutors implemented and no knowledge engineering tools developed. The MATHESIS framework provides both general, domain-independent knowledge engineering tools and a knowledge engineering language as well as special authoring tools for model-tracing tutors.

ASPIRE[5] is a knowledge engineering framework for the development of constrained-based tutors (Mitrovic, Martin, Suraweera, Zakharov, Milik, & Hooland, 2009). These tutors do not use a domain model to trace the student's solution in a step-by-step basis, but they are equipped with a set of constraints that describe the forms of correct solution(s) for the tutored problem. In a comparative study between model-tracing and constraint-based tutors (Mitrovich, Koedinger, & Martin, 2003), the authors conclude that "*Model-tracing is an excellent choice for domains where appropriate problem solving strategies are well-defined, and where comprehensive feedback on them is desirable. On the other hand, CBM offers a workable alternative when such strategies are not available or appropriate, or there is too little time or resources to build a model-tracing knowledge base*". Therefore, in addition to the breadth and depth issue, constraint-based tutors cannot provide the granularity necessary for, e.g., an algebra tutor.

Whenever there is need for a broad and/or deep domain model, authors (knowledge engineers) usually start from scratch and fall back to customized solutions. Two such examples are the Andes[6] physics tutor (VanLehn, Lynch, Schulze, Shapiro, Shelby, Taylor, Treacy, Weinstein, & Wintersgill, 2005) and the Visual Classification Tutoring Framework (VCT) (Crowley & Medvedeva, 2006).

Andes contains 356 physics problems (mechanics, electricity and magnetism) solved by a knowledge base of 550 physics rules. These rules comprise "major principles", like Newton's second law ($F = m \cdot a$), as well as "minor principles", like mathematical and common sense justifiers. The creation and maintenance of such a broad, deep and granular domain model raises drastically the demands in expertise and time resources (VanLehn et al., 2005). As far as it concerns the development time, Andes itself took five years to be built, while its development was based on the Cascade (VanLehn, 1999) and Olae (VanLehn, Johnes, & Chi, 1992) projects. Finally, there were significant scalability problems, since in order to add a new rule to the domain model knowledge engineers should re-inspect the whole model. (VanLehn et al., 2005)

The same findings hold for the Visual Classification Tutoring (VCT) framework, which generally supports the development of tutors for visual classification, but specialises in medical domains like radiology, haematology and pathology. The framework makes the best provision for accommodating broad, deep, granular and scalable domain models by using ontologies to represent separately generic models for the domain model, the task model and the pedagogical model. This generic framework was used to develop SlideTutor[7], a model-tracing tutor for a sub-domain of inflammatory diseases of skin, covering 33 diseases with 50 different diagnostic features. Once again, the expertise and time costs are high: an expert pathologist in cooperation with a knowledge engineer must annotate each diagnostic case with the contained disease and its diagnostic features. Based on this information, the task model produces dynamic solution graphs that guide the student in his/her diagnosis.

From the description of these two systems, Andes and VCT, it becomes clear that the development of model-tracing tutors with broad and deep domain expertise models without the use

[4] http://astus.usherbrooke.ca
[5] http://aspire.cosc.canterbury.ac.nz
[6] http://www.andestutor.org/

of authoring tools raises significantly the threshold both in human expertise and time for their maintenance and further development.

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 and facilitate the reuse of content and tools (Dicheva, Mizoguchi, & Greer, 2009). Examples of intelligent tutoring systems that use ontologies are *Activemath* (Melis, Andrès, Büdenbender, Frischauf, Goguadze, Libbrecht et al. 2001), which uses ontological representation of mathematical concepts, learning goals and acquired knowledge, and *SlideTutor*[7]. However, these are intelligent tutoring systems and not knowledge engineering systems.

An ontology-based knowledge engineering system for constraint-based tutors is ASPIRE (Suraweera et al., 2009). It uses ontologies to define the concepts of the domain and then, based on these definitions, it provides the constraints for possible solutions used by the authored constraint-based tutors.

Another related line of research has to do with ontology authoring systems that support Controlled Natural English (CNL), like ROO (Denaux, Dolbear, Hart, Dimitrova, & Cohn, 2011) and AceView (Kaljurand, 2008). The purpose of these systems is to involve domain experts in the development of ontologies using CNL to describe their conceptual knowledge. Although ROO aims at lowering the expertise threshold for domain experts and improves knowledge reuse, there are two main differences with the MATHESIS framework: a) ROO aims at the ontological representation of the *static* part of the domain language, namely the concepts, their properties and their relationships, while MATHESIS covers both *static* and *procedural* domain knowledge and b) In ROO, domain experts are involved in the initial phase of the ontology development cycle while knowledge engineers follow to validate and amend the developed ontology; in MATHESIS, knowledge engineers build executable knowledge engineering models using a specialised language, ONTOMATH (see Section 4), and these models guide domain experts in building a complete knowledge based system (ITS).

The most relevant work to the MATHESIS framework is the OMNIBUS/SMARTIES project (Mizoguchi, Hayasi, & Bourdeau, 2009). 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 modelling environment and guidelines for developing 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 domain and knowledge engineering expertise of any kind of tutor 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.

Although ASPIRE and OMNIBUS/SMARTIES are ontology-based knowledge engineering systems, they differ from the MATHESIS framework which constitutes a meta-knowledge engineering system. These systems provide specific knowledge engineering programs that use a *static* ontological representation of tutoring and knowledge engineering expertise to build tutors of a *specific* kind. The MATHESIS framework provides meta-knowledge engineering tools and a knowledge engineering language for expert knowledge engineers to write knowledge engineering programs in the form of executable OWL-S knowledge engineering processes. These knowledge engineering programs can then be *executed* by the knowledge engineering tools to guide less expert knowledge engineers (e.g., domain experts) in generating the ontological representation of *any* kind of knowledge-based system (tutor). This ontological representation can then be translated to program code.

Our approach combines the research in the field of authoring tools for ITSs with the field of knowledge engineering tools for knowledge-based systems. This line of research starts with the first attempts to define reusable problem-solving knowledge through the introduction of the concepts of *Generic Tasks* (Chandrasekaran, 1986) and *heuristic classification* (Clancey, 1985). It continues with the concepts of *task ontologies* (Mizoguchi, Vanwelkenhuesen, & Ikeda, 1995), the development of knowledge modelling frameworks like the MULTIS project (Mizoguchi,

[7] http://slidetutor.upmc.edu/

Vanwelkenhuesen, & Ikeda, 1995), the Protégé project (Puerta & Musen, 1992), the KADS (Wielinga, Schreiber, & Breuker, 1992) and CommonKADS (Schreiber, et al., 1999) projects. The latter introduced the concept of Problem Solving Methods (PSMs). With the emergence of the Web, the necessity for representing and deploying PSMs in a shareable and reusable way led to their semantic (ontological) representation as Web Services. The ultimate goal is the development of knowledge-based systems from reusable knowledge components found in the web, a task known as *automated web service composition*. Various frameworks with web services description languages have been developed, OWL-S being one of them. Although it is not our immediate intention to view ITS knowledge engineering as a web service composition task, we set the foundations, focusing on the shareability and reusability of knowledge engineering and tutoring knowledge provided by OWL-S.

2.2. Ontological Engineering and the Knowledge Gap Problem

In an extensive survey of authoring tools, Murray (2003a) concluded that they suffered from a number of problems such as isolation, fragmentation and lack of communication, interoperability and re-usability of the tutors they build. The same problems had been identified three years earlier in (Mizoguchi & Bourdeau, 2000). These problems are not specific to the domain of ITS authoring, as they penetrate the whole area of knowledge-based systems development (Lenat & Guha, 1990; Lenat, 1995). A highly promising solution to all of them is *ontological engineering*, that is, the development of ontologies that represent declaratively the expertise that lies inside any intelligent system (Mizoguchi, 2004). The main advantages of the use of ontologies are that:

1. they impose a systematic and structured development of knowledge, just like developing a mathematical theory with definitions, properties, axioms and theories; and
2. the developed knowledge being in a declarative form is open for inspection and therefore mostly reusable (Gómez-Pérez, Fernández-López, Corcho, 2004).

The main goal of the MATHESIS project, presented in this article, is to develop knowledge engineering tools for model-tracing tutors in mathematics. Based on the success of the ontological engineering approach in the domain of expert systems (Aitken & Sklavakis, 1999; Lenat, 1995; Sklavakis, 1998), as well as in the domain of intelligent tutoring systems (Mizoguchi, Hayashi, & Bourdeau, 2009), we set two research goals:

1. the complete ontological representation of a model-tracing tutor's modules, that is, the tutoring model, the domain expertise model, the student model, the user interface, as well as of the knowledge engineering expertise that was used to build these models; and
2. the extensive use of standardized languages and publicly available modular tools.

For these reasons, we adopted a bottom-up approach: Initially, an Algebra Tutor was developed to be used as a prototype target tutor (Sklavakis & Refanidis, 2008; Sklavakis & Refanidis, 2013). This tutor has a domain model of considerable breadth, depth and granularity, easily scalable. Then, based on the knowledge used to develop the Algebra Tutor, an initial version of the MATHESIS ontology has been developed using the Ontology Web Language - OWL[8] (Sklavakis, & Refanidis, 2010). As this first version of the ontology was developed in a bottom-up direction, it emphasized on the representation of the tutor's models, namely the interface, tutoring and domain expertise models. The ontology also contained a representation of the knowledge engineering expertise at a rather conceptual level. At the final stage of the project, the generic meta-knowledge engineering tools were developed. These tools include:

[8]http://www.w3.org/TR/owl-features/

1. An executable knowledge engineering language, ONTOMATH, based on the process model of OWL-S[9];
2. meta-knowledge engineering tools for the development of ONTOMATH executable knowledge engineering expertise models, that is, an ontological representation of the declarative and procedural knowledge engineering expertise; and
3. an interpreter for executing the ONTOMATH knowledge engineering models.

Using these tools, we have built a knowledge engineering model that, when executed, builds the ontological representation of a model-tracing monomial multiplication tutor identical to the one contained in the original Algebra Tutor. In parallel, we developed knowledge engineering tools for the development of model-tracing tutors. These tools are used to support the meta-knowledge engineering tools in the development of the executable knowledge engineering model by automating some top-level knowledge engineering processes of the model-tracing tutor under development and providing visualisation and browsing facilities for the inspection of the tutor's developed models.

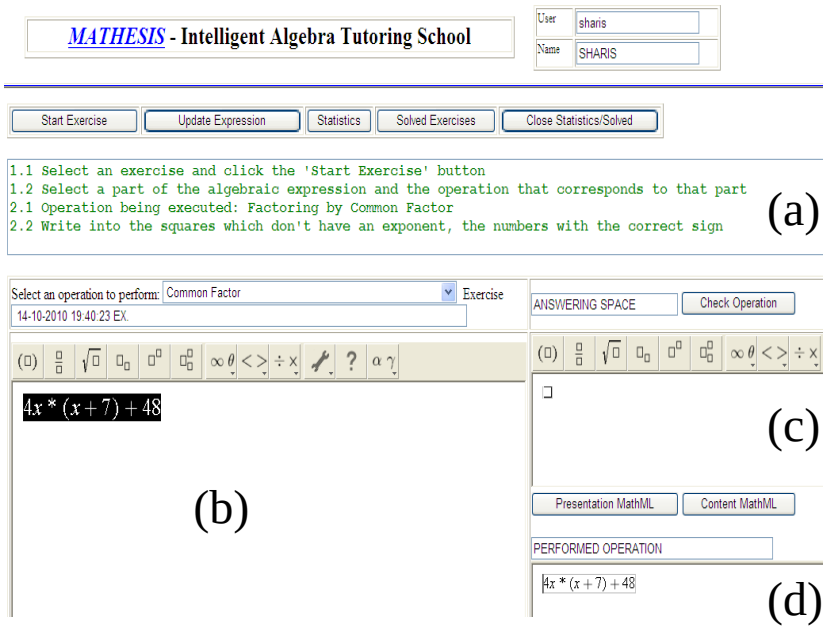


Fig. 1. The MATHESIS Algebra Tutor Interface: (a) The messages area, (b) The algebraic expression area, (c) The student answering area and (d) The performed operation area

3. The MATHESIS Algebra Tutor

The MATHESIS model-tracing Algebra Tutor[10] (Sklavakis & Refanidis, 2013) was developed as a prototype target tutor for the MATHESIS project, having knowledge reuse as its primary design guidelines. Furthermore, the architecture of the system was based in open, standardized and modular representations. The fulfilment of these requirements led us to implement the tutor using HTML for the user interface and JavaScript for the domain expertise and tutoring models. The user interface has four main parts: the messages area, the algebraic expression area, the student's answering area and the performed operation area (Figure 1). The primary interface element is Design Science's WebEq Input Control applet[11], an editor for displaying and editing mathematical expressions. There are three such input controls: the algebraic expression, the answering space and the performed operation input controls. WebEq Input control is scriptable through JavaScript and

[9] http://www.w3.org/Submission/OWL-S/
[10] http://users.sch.gr/dsklavakis/mathesis/en/MATHESIS_Main_Frameset.htm
[11] http://www.dessci.com/

uses MathML[12] to represent algebraic expressions. So, during the problem solving process, the problem solving state as well as the student solution steps are represented via the open MathML standard and, therefore, they are *interoperable*, that is, *inspectable*, *recordable* and *scriptable* (Murray 2003).

The development of the domain expertise model was based on deep cognitive task analysis (Anderson, Corbett, Koedinger, & Pelletier 1995). The top-level math skills that the tutor covers are monomial multiplication, division and power; monomial-polynomial and polynomial-polynomial multiplication; parentheses elimination; collect like terms; identities (square of sum and difference, product of sum-difference, cube of sum and difference); and factoring (common factor, identities, trinomial). Each one of these 13 top-level math skills is further analysed in more detailed sub-skills leading to a fine grained domain model of 104 primitive math skills. As an example we consider the *multiply-monomials* skill. This is decomposed in two sub-skills, *multiply-coefficients* and *multiply-mainParts*. The *multiply-mainParts* is further decomposed in primitive math skills like finding common variables, adding their exponents, finding non common variables and copying their exponents. This decomposition is implemented through JavaScript functions that correspond to the production rules, and JavaScript data structures (simple variables, arrays, custom objects) that correspond to the facts of a rule-based system. There are also relevant functions for common error checking like omitting variables, or not adding the exponents of common variables.

Based on this broad, deep and granular domain model, the tutor's tutoring model uses *deep model tracing with intelligent task recognition*. The tutor uses *intelligent parsing* of the MathML representation of the algebraic expression and records which tasks (algebraic operations) are present, their order and their operands. Then, it asks from the student to select a part of the expression and suggest the operation that must be performed. If the student is wrong, the tutor provides feedback about the correct operation. Otherwise, the tutor guides the student in performing the operation step-by-step. In each step, the tutor checks the student's answers and compares them with its domain model to provide feedback (*model tracing*). If a task contains subtasks, e.g. a polynomial multiplication contains monomial multiplications, the tutor guides the student to perform these subtasks in the same step-by-step manner (*deep model tracing*).

It is exactly these features, broad and deep domain model, intelligent task recognition and deep model tracing, that deal directly with the scaling-up problem: the MATHESIS tutor can handle *any* algebraic expression containing *any* combination of the math tasks described above. Thus, the MATHESIS tutor can guide a student in expanding expressions like $(2x+3)^2 - 2(2x+3)(2x-3) + (2x-3)^2$ or factor expressions like $4(x-1)^2 - 9(x-2)^2$.

4. Overview of the MATHESIS meta-knowledge engineering framework

The MATHESIS framework is mainly a meta-knowledge engineering framework. It is well known that knowledge engineering (KE) is knowledge of how to extract problem-solving knowledge from domain experts, represent this knowledge in a suitable format and implement a system that uses this knowledge to solve problems like a human expert (Aitken & Sklavakis, 1999; Lenat, 1995; Sklavakis, 1998).

In the case of KE (authoring) systems for ITSs, a meta-KE framework should enable (meta-) knowledge engineers to extract related knowledge from expert ITS knowledge engineers (authors), that is, cognitive scientists and programmers (AI or general purpose); represent this knowledge in a suitable format; and implement a system that uses this knowledge to guide "knowledge engineers" of lower levels of expertise (domain experts) to build knowledge-based (ITS) systems. To achieve these three objectives, the MATHESIS meta-KE framework adds a *semantic level* on top of the knowledge level of each KE (authoring) framework (Figure 2). Its purpose is to represent declaratively (ontologically) the KE expertise used to build knowledge-based systems (ITS), now lying partially unexpressed into the minds of KE experts and partially expressed into the KE tools, as well as the developed domain problem-solving (tutoring) knowledge hard-wired into the knowledge-based systems (ITSs) themselves.

[12] http://www.w3.org/Math/

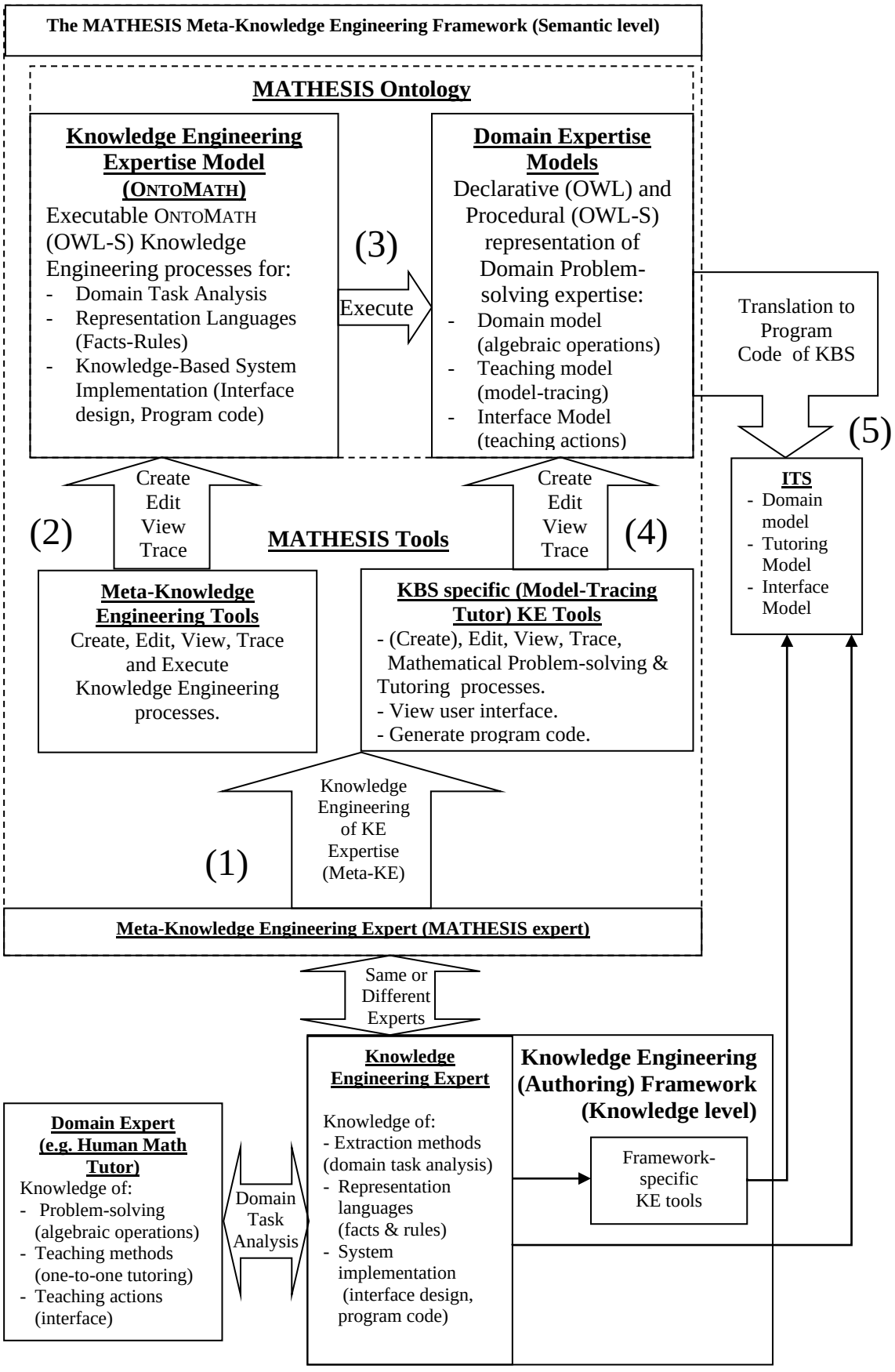


Fig. 2. The MATHESIS Meta-Knowledge Engineering Framework

The key point of the proposed framework is the *ontological declarative representation* of these two kinds of knowledge. At the same time, and this was the most challenging problem, these declarative representations should also be *executable*. More specifically, the deployment of the framework is done in the following stages (Figure 2, bottom to top):

1. A knowledge engineer specialized in the MATHESIS framework (meta-knowledge engineer), extracts the expertise from the domain KE experts (ITS authors), cognitive scientists and AI programmers in the case of ITSs. The KE expertise must cover all stages of knowledge-based system (ITS) development, that is, analysis, design and implementation (see Section 6). This constitutes a crucial difference between the framework-specific KE tools and the objectives of the MATHESIS framework: the former support parts of the KBS (ITS) development stages, usually leaving out the most difficult ones like the domain analysis stage, while the latter allows meta-knowledge engineers to encode KE expertise of any stage. It must be noted that this phase requires the development of knowledge about knowledge engineering (meta-knowledge). The MATHESIS framework does not provide any specific methodology for this kind of meta-knowledge engineering. In principle, any of the existing KE methodologies could be used such as the KADS and CommonKADS ones (Kingston, 1995).
2. Using the meta-KE tools (Figure 3b), the meta-knowledge engineer creates an executable ontological model of the extracted KE expertise, the *knowledge engineering expertise model*. This model contains KE processes described in ONTOMATH, a special purpose language developed within the framework (Section 5.2). ONTOMATH defines two kinds of KE processes: (a) *composite KE processes*, which correspond to the functions/procedures of a programming language and are represented using the process model of OWL-S, and (b) *atomic KE processes*, which correspond to the statements of a programming language.
3. When the KE processes are executed by a non-expert knowledge engineer (e.g., domain expert), the ONTOMATH interpreter executes them by calling corresponding Java methods which in turn use the Protégé API to guide the domain expert in building the ontological representation of the KBS models – conceptual (Section 6.1), problem-solving (Section 6.2), tutoring (Section 6.3), in the case of ITS - into the MATHESIS ontology (Figure 3c). It also builds the models of the KBS program code (Section 6.4) and interface (Section 6.5). Therefore, the KE processes are the semantic representation of the domain-specific knowledge engineering tools. The ontological representation of the KBS's (ITS's) various models (cognitive, teaching, interface) contain both declarative and procedural domain knowledge. An example of declarative domain knowledge would be the interface structure (interface model) or the problem-solving concepts and stages of the domain problem-solving model. An example of procedural knowledge would be the model-tracing algorithm (tutoring model) or the problem-solving steps of the domain model. In the MATHESIS framework these knowledge elements are defined by the meta-knowledge engineer as generic elements. Declarative domain knowledge elements are defined using the common OWL structures: classes, instances, properties and values. Procedural domain knowledge elements are defined using the process model of OWL-S, just like the composite KE processes described in stage 2. It is these generic knowledge elements that the executed KE processes act on, guiding the domain expert to create specific-ones for the KBS (ITS) under development.
4. The meta-knowledge engineer may develop KBS-specific (e.g., Model-tracing Tutor) tools to help himself develop the KE model and the domain experts in developing the KBS (ITS). These are mainly visualisation tools, although they can also provide manual creation and editing of KBS-specific knowledge elements based on generic ones. This last facility aims at accommodating more knowledge engineers that can develop parts of the KBS (ITS) directly, without executing the corresponding KE processes. We have developed a suite of such KBS-specific tools for model-tracing tutors (Figure 3a).
5. Having created the ontological representation of the tutor, the domain expert can create its implementation by translating the ontological model to specific programming languages. For example, in the case of the MATHESIS Algebra Tutor, the interface model is translated to HTML and the domain and teaching models to JavaScript. These translations

are performed automatically by special translation tools. In case of other target programming languages, we need to develop their corresponding ontological representation as well as the translation tool.

All of the aforementioned stages are performed using the MATHESIS tools (Figure 3).

5. Procedural knowledge representation

The main component of the MATHESIS framework is the Ontology. It contains three kinds of knowledge:

1. The declarative knowledge of the tutor, such as the interface structure and the problem-solving concepts and stages of the domain model,
2. the procedural knowledge of the KBS, such as the teaching and math domain expertise models of a model-tracing tutor and, finally,
3. the knowledge engineering expertise, that is, the declarative and procedural knowledge that is needed to develop the tutor.

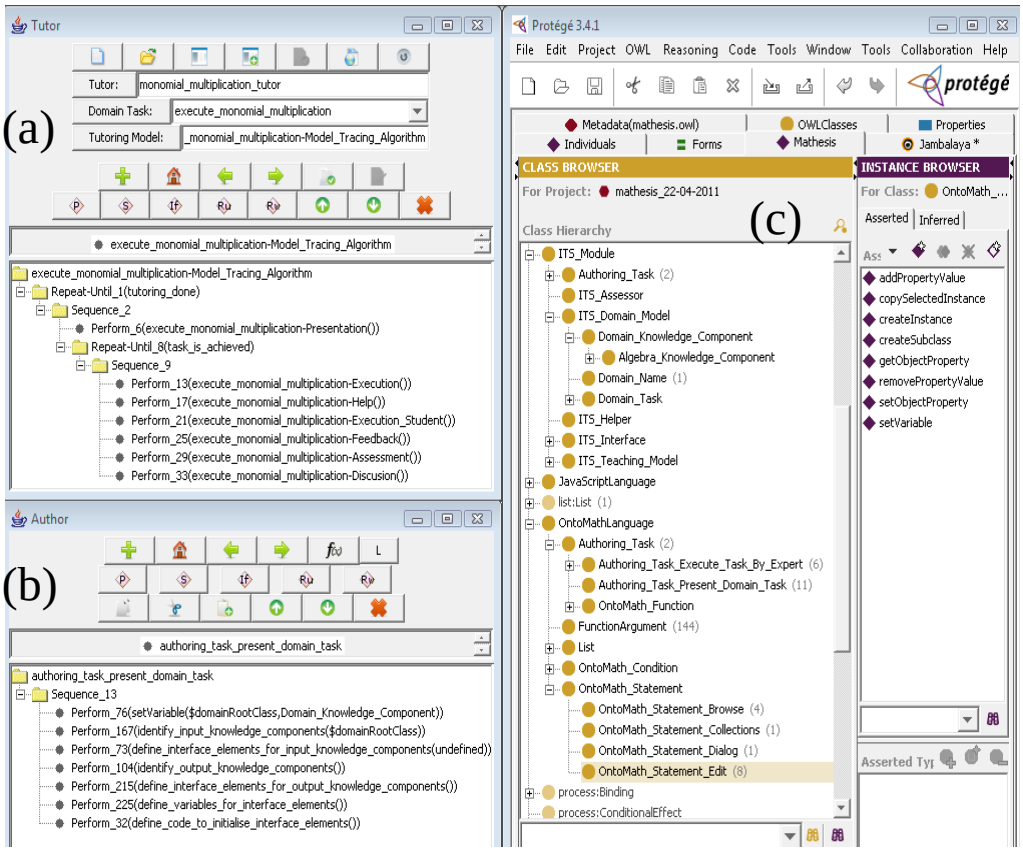


Fig. 3. The MATHESIS Tools as a tab widget in Protégé:
(a) KBS-specific (Model-tracing Tutor) Knowledge Engineering Tools, (b) Knowledge Engineering Processes (Meta-Knowledge Engineering) Tools, (c) The MATHESIS Ontology Tab

While the declarative knowledge is represented with the basic OWL components, the procedural knowledge, both domain and knowledge engineering, is represented via the *process model* of the OWL-S web services description ontology. Through OWL-S, every knowledge engineering or domain task is represented as a knowledge engineering or domain task process, composite or atomic.

Using the OWL-S process model to represent ontologically procedural knowledge, like teaching, math problem-solving or KE expertise is the key advantage of the MATHESIS framework that gives a new perspective in the development of reusable KE expertise for

Knowledge Based Systems. In the following, we start by briefly presenting the OWL-S process model, before delving into the details of its application to our case.

5.1. The OWL-S process model

OWL-S is a web service description ontology designed to enable the following tasks:

- Automated discovery of Web services that can provide a particular class of service capabilities, while adhering to some client-specified constraints.
- Automated Web service invocation by a computer program or agent, given only a declarative description of the service.
- Automated Web service selection, composition and interoperation to perform some complex task, given a high-level description of an objective.

The last task is of interest for the MATHESIS framework and therefore we focus on it. To support this task, OWL-S provides, among other things, a language for describing service compositions as seen in Figure 4 (Martin et al., 2005)

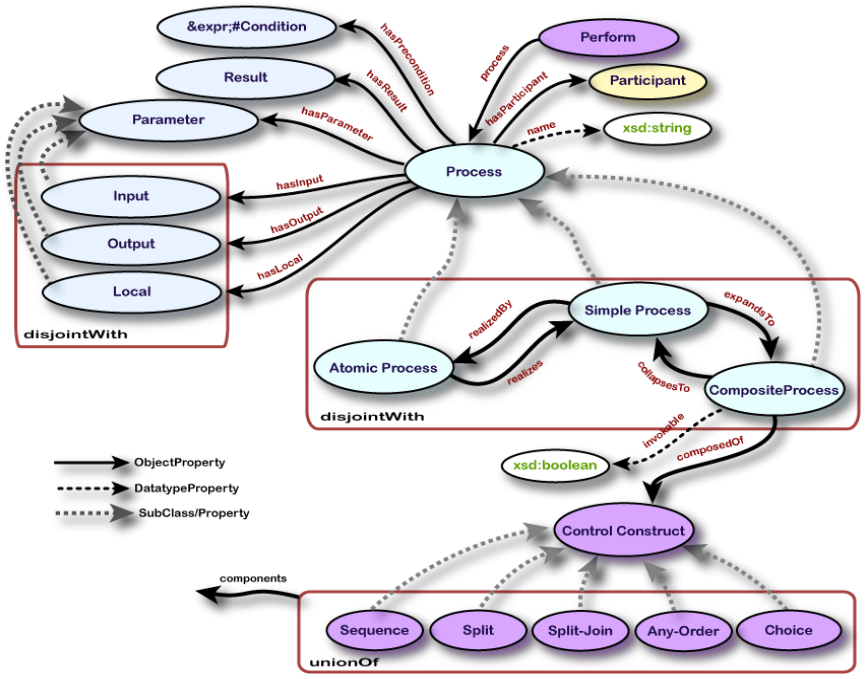


Fig. 4. Top level of the OWL-S *process* ontology (from Martin et al., 2005)

Every service is viewed as a *process*. OWL-S defines Process as a subclass of ServiceModel. There are three subclasses of Process, namely the AtomicProcess, CompositeProcess and SimpleProcess. Atomic processes correspond to the actions a service can perform by engaging it in a single interaction. In the MATHESIS ontology they represent simple statements that perform either domain or KE elementary tasks. Composite processes correspond to tasks that require multi-step actions. In the MATHESIS ontology they represent functions, either domain or knowledge engineering, that call other functions (composite processes). Finally, simple processes provide an abstraction mechanism to provide multiple views of the same process. Currently, they are not used in the MATHESIS framework. Composite processes are decomposable into other composite or atomic processes. Their decomposition is achieved by using control constructs such as Sequence or If-Then-Else. Table 1 shows the most common control constructs supported by OWL-S.

Any composite process can be considered as a tree whose non-terminal nodes are labelled with control constructs. The leaves of the tree are invocations of other processes, composite or atomic.

These invocations are indicated as instances of the Perform control construct. This special control construct takes as a parameter a process, either composite or atomic. In the MATHESIS framework a Perform with an atomic process corresponds to the execution of a statement, whereas a Perform with a composite process corresponds to calling a function. This tree-like representation of composite processes is the key characteristic of the OWL-S process model and has been used in the MATHESIS Ontology to represent both knowledge engineering and domain task procedural knowledge

Table 1. Common control constructs supported by the OWL-S process model

Control Construct	Description
Sequence	A list of control constructs to be performed in order
Choice	Calls for the execution of a single construct from a given bag of control constructs (given by the components property). Any of the given constructs may be chosen for execution
If-Then-Else	It has properties ifCondition, then and else holding different aspects of the If-Then-Else construct
Repeat-While & Repeat-Until	The initiation, termination or maintenance condition is specified with a whileCondition or an untilCondition respectively. The operation of the constructs follows the familiar programming language conventions.

5.2. Procedural knowledge engineering expertise representation

The MATHESIS framework allows expert meta-Knowledge Engineers to capture the whole KE effort by providing an executable KE model building language, namely ONTOMATH. In ONTOMATH, each KE task is represented as a KE process, either composite or atomic (Figure 5).

Composite KE processes correspond to functions of a programming language that can be called, get and return values. This is achieved by two means: a) Each composite process has a property, hasFormalParameters, which keeps a list of the process formal parameters, and b) each Perform has a property, hasRealParameters, which keeps the list of the parameters at call time. During execution of a Perform construct by the meta-KE tools (Figure 3b), the interpreter matches the values of the real parameters to those of formal parameters. The values of the two properties, hasFormalParameters and hasRealParameters, are defined by the meta-knowledge engineer in the ontology with the help of the meta-KE tools (Figure 3b). The recursive analysis of composite KE processes ends to atomic KE processes, which are instances of the OntoMathStatement, a subclass of AtomicProcess. Each OntoMathStatement instance corresponds to an operation that must be performed to the MATHESIS Ontology (see Table A1, Appendix).

ONTOMATH KE processes, composite and atomic, are classified in classes and subclasses according to the kind of KE expertise they represent and the part of the KE endeavour they implement. For example, in Figure 5, there are ONTOMATH processes (identify-input-knowledge-components, define-interface-elements-for-input-knowledge-components, define-variables-for-interface-elements, define-code-to-initialise-interface-elements) for implementing the task of defining how the math problem will be presented to the student, which is a tutoring (domain) task (class Authoring-Task-Present-Domain-Task). Other ONTOMATH processes, like get-HTML-Element-Property, serve the task of HTML programming (class Programming-Task-HTML). Others, like get-interface-element-reference, serve the task of JavaScript programming (class Programming-Task-JavaScript). As a consequence, the ONTOMATH language is not only capturing KE expertise but, at the same time, is *classifying* this expertise. This way it enables reasoning on that knowledge, that is, discovery, retrieval, reuse and modification, by knowledge engineers of different expertise levels.

ONTOMATH statements are *grounded* to actual Java program code. When the MATHESIS meta-KE tools interpret a Perform construct that calls an ONTOMATH statement, they execute its

corresponding Java code, which performs the statement's operations on the ontology that represents the KBS (tutor) under development. It must be noted that the set of ONTOMATH statements is not fixed. Expert meta-knowledge engineers can define their own atomic KE (ONTOMATH) statements by: a) using the meta-KE tools to define in the MATHESIS Ontology the values of property hasFormalParameters for the new statement and b) write 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 meta-KE tools leads to the creation of the KBS's (tutor's) ontological representation and, consequently, to the implementation of the KBS (tutor).Therefore, the ONTOMATH KE processes form an ontological representation of a *meta-program* that handles the ontological representation of the tutor as its data.

.

Fig. 5. Part of the ONTOMATH Knowledge Engineering Processes Ontology

6. Using the MATHESIS framework

In this Section we describe how meta-knowledge engineers create a KE expertise model using as an example a model-tracing tutor for monomial multiplication, as well as how domain experts can create the monomial multiplication tutor by executing the KE model. The meta-KE model for the monomial multiplication tutor was based on the expertise used to create the monomial multiplication part of the MATHESIS Algebra Tutor. Provision has been taken so that both the KE model and the tutor's model can be extended for meta-knowledge engineering all the math skills tutored by the MATHESIS Algebra Tutor.

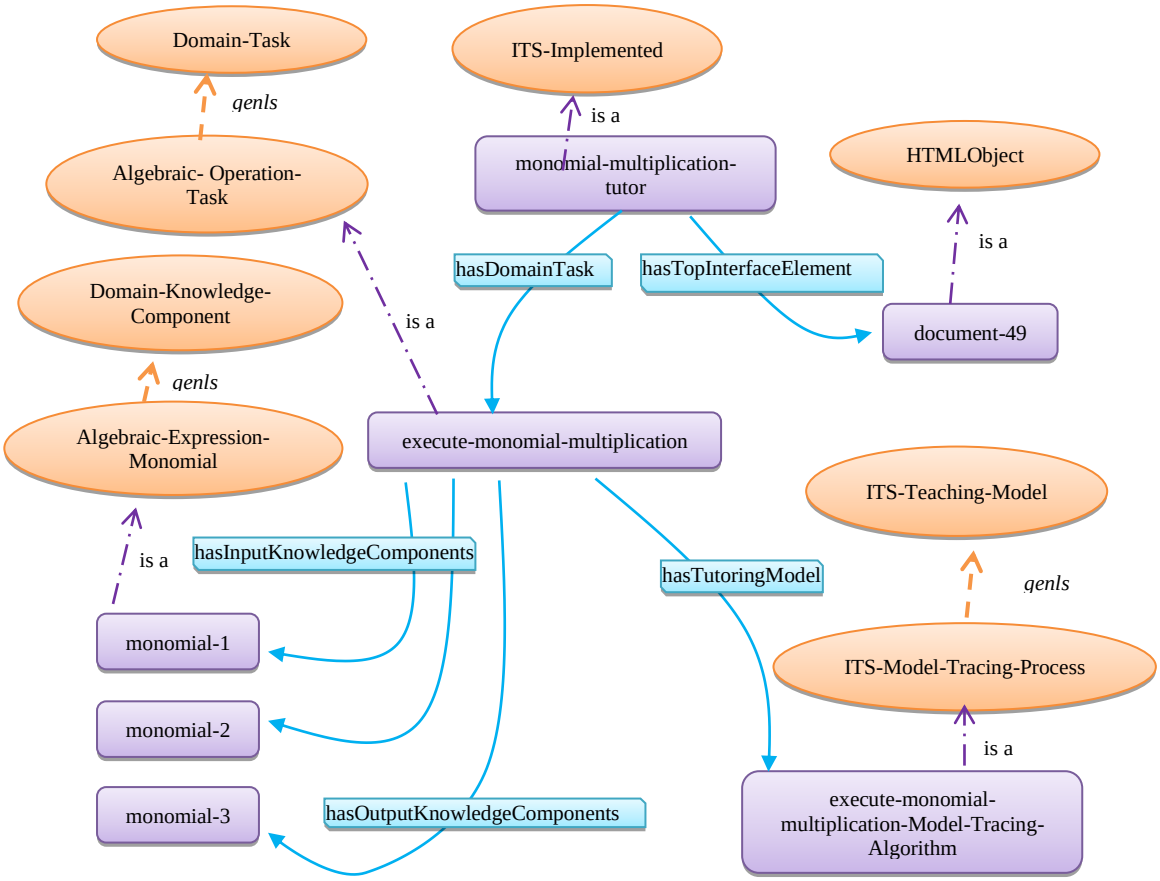


Fig. 6. The top-level ontological representation of the tutor created by the domain expert

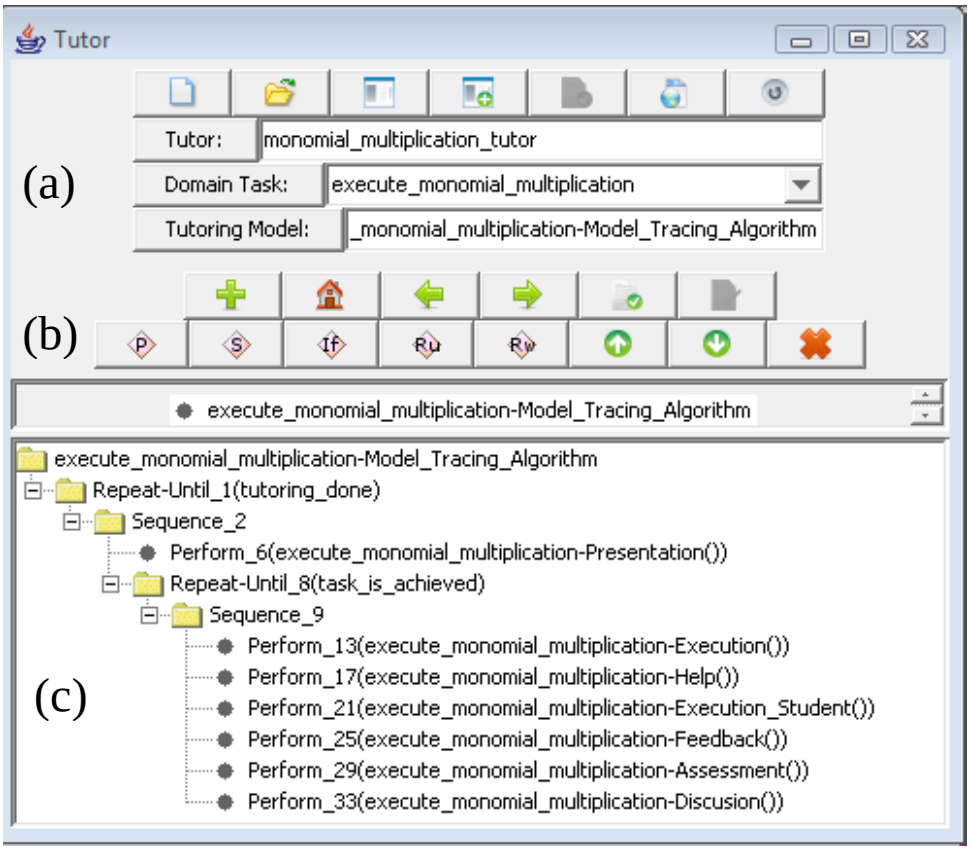
6.1 Tutor Initialization[13]

The first KE task is to define an instance of the tutor (KBS) in the Ontology. At the top level of the MATHESIS Ontology, every tutor is represented as an instance of class ITS-Implemented. In Figure 6 this instance is monomial_multiplication_tutor. The domain expert creates this instance once in the first KE session; in subsequent sessions the domain expert selects the tutor to edit. In both cases, the tutor Initialization KE tools (Figure 7a) automatically select the ITS_Implemented class in the MATHESIS ontology tab (Figure 3c). Class ITS-Implemented is a top class created by the meta-knowledge engineer and can have subclasses like ITS-Implemented_Algebra, which are used for classifying the various tutors. This classification can use various criteria defined by the meta knowledge engineer, such as the tutors' domain (math, physics, programming), thus making easier for domain experts to locate a specific tutor in the ontology.

6.2. Domain Problem-Solving Expertise model

The next KE task is to define a mathematical domain task that the tutor teaches, e.g., monomial multiplication. The domain expert can add in the ontology new tasks or select from existing ones. This KE task is also performed using the Tutor Initialization Tools (Figure 7a). The created or selected instance of the mathematical domain task is added to property hasDomainTask that keeps a list of the tasks (here execute-monomial-multiplication) that the tutor teaches (Domain-Task instances). Figure 6 shows the domain task instance execute-monomial-multiplication as a value of the hasDomainTask object property. The domain expert implements later the domain tasks by executing the KE processes developed by the meta-knowledge engineer according to each domain task. As with the tutor instances, domain task instances are classified in a hierarchy with Domain-Task being the root (Figure 6). Notice that task execute-monomial-multiplication is actually an instance of the Algebraic-Operation-Task subclass.

[13] http://ai.uom.gr/dsklavakis/en/mathesis/kes2011/01-Authorring_Tools.mp4



.

Fig. 7. The KBS domain-specific (Model-Tracing Tutor) Knowledge Engineering Tools: (a) The Tutor Initialization Tools, (b) The Advanced Knowledge Engineering Tools for Tutoring Domain Processes , (c) Tree representation of the tutoring domain task process Model-Tracing-Algorithm adapted for the execute-monomial-multiplication domain task.

For each math domain task, the domain expert must define in the ontology the domain concepts given for the task and the domain concepts asked for the task (knowledge components). For the execute-monomial-multiplication task the input knowledge components are two monomials and the output knowledge component is their product, a monomial too. Domain concepts' instances are classified in a hierarchy with class Domain-Knowledge-Component on top (Figure 6, middle left). In the Ontology, the given and asked domain concepts of the domain task are kept by properties hasInputKnowledgeComponents and hasOutputKnowledgeComponents respectively (Figure 6, bottom left). In Figure 6 there are three instances: monomial-1, monomial-2 and monomial-3. When a domain expert needs to define in a tutor the domain concept of monomial, he/she has to clone the *generic instance* monomial created by the meta-knowledge engineer. The domain expert creates these clones by executing the appropriate KE processes created by the meta-knowledge engineer. Process identify_input_knowledge_components (Figure A1, Appendix) guides the domain expert in creating new instances of domain concepts that form the input for a domain task. In the case of the monomial multiplication tutor the domain task is execute-monomial-multiplication and the input domain concepts that must be created are two monomials, namely monomial-1 and monomial-2. Programming-savvy readers can follow the ONTOMATH code using the definitions of the ONTOMATH statements given in Table A1 (Appendix). Therefore, the state of the ontology shown in Figure 6 is after the domain expert has executed the appropriate KE processes that guided him to create the three specific instances, monomial-1, monomial-2 and monomial-3, by cloning monomial.

6.3.Tutoring model

The top level representation of the tutor's procedural knowledge in the ontology is the model-tracing algorithm represented as a *generic composite tutoring (domain task) process*, named ModelTracingAlgorithm. This algorithm is an implementation of the two-loop structure of intelligent

tutoring systems described in (VanLehn, 2006) and was developed by the meta- knowledge engineer using the Advanced KE Tools for Tutoring (Domain) Processes (Figure 7b). The tools allow meta-knowledge engineers and advanced domain experts to create parts of the tutor's procedural domain knowledge (tutoring processes) directly, without executing KE processes. This ability is necessary for complicated tutoring (domain task) processes, like the ModelTracingAlgorithm, which demand high expertise and must be provided to domain experts as libraries, that is, *generic tutoring (domain task) processes* in the MATHESIS framework terminology. When the domain expert selects the ModelTracingAlgorithm as the tutoring model, the Tutor Initialization KE Tools (Figure 7a) copy its structure and create a new instance, execute-monomial-multiplication-Model-Tracing-Algorithm (Figure 7c), for the execute_monomial_multiplication domain task. Consequently, the meta-knowledge engineer can define any number of generic tutoring (domain task) models (e.g., model-tracing, example-tracing, constrained-based, other) ready to be selected by the domain expert and adapted by the execution of KE processes for the specific domain task.

The tree structure of the process, adapted and displayed by the Tutor Initialization Authoring tools for the execute-monomial-multiplication task, is shown in Figure 7c. Each step of the algorithm is a top level tutorial (domain task) action. Each of these steps is also a *composite tutoring (domain task) process* that analyses the tutorial steps further down to more simple ones like giving a hint, showing an example, recalling math formulas or rules and so on. The domain expert develops these tutoring (domain) sub-tasks by executing corresponding KE processes created by the meta-knowledge engineer. Each tutoring (domain task) process is associated with a KE process by the meta-knowledge engineer via its hasAuthoringProcess property. The code for tutoring process execute-monomial-multiplication-Presentation (Figure 7c, Perform 6) is created by the execution of the KE process authoring-task-present-domain-task displayed by the meta-KE tools (Figure 8). The domain expert is guided to perform the following KE sub-tasks:

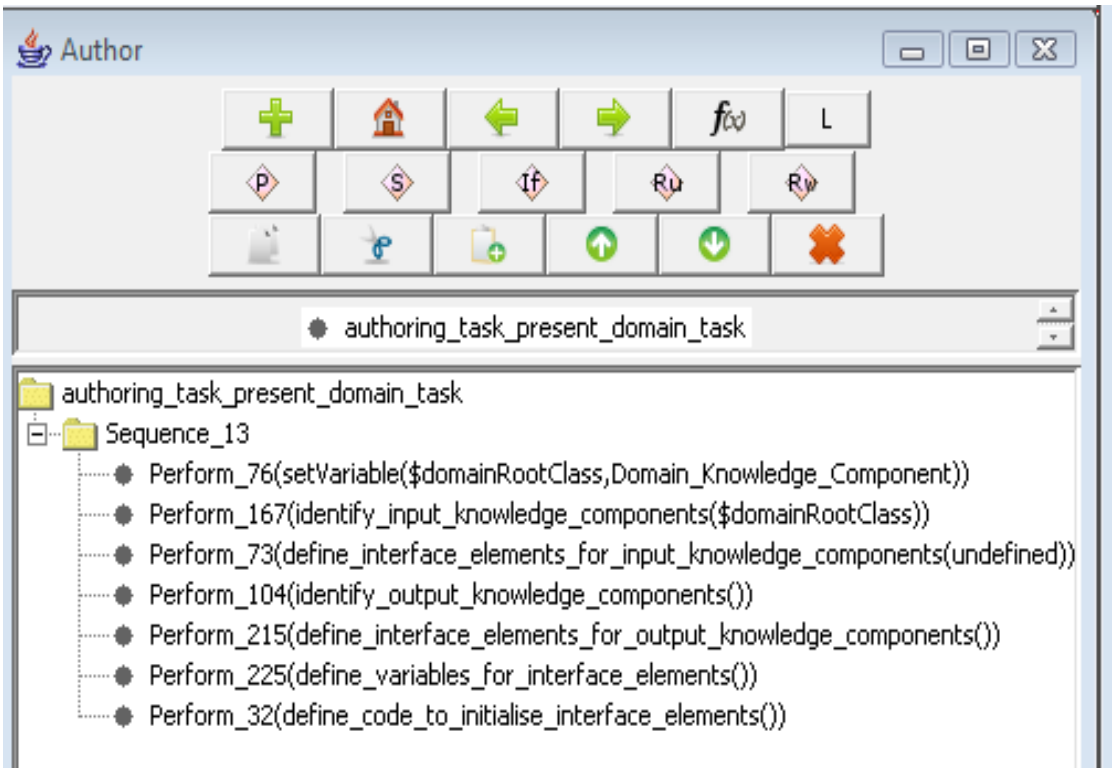


Fig. 8. The Meta-Knowledge Engineering (ONTOMATH) Tools displaying the Knowledge Engineering (ONTOMATH) process authoring-task-present-domain-task. This process creates the Tutoring (Domain) Process execute-monomial-multiplication-Presentation which presents the initial problem state of a (monomial multiplication) tutor.

1. Identify the input domain concepts (monomials) of the domain task.
2. Define the interface elements (WebEq_Input_Control applets) for the input domain concepts (monomials).
3. Repeat the aforementioned steps for the task's output concepts, that is, a monomial holding the product.
4. Define the JavaScript variables that hold the references of the WebEq_Input_Control applets.
5. Define the JavaScript code that initializes the variables referencing the WebEq_Input_Control applets

6.4. Program code model

In programming terms, the ModelTracingAlgorithm composite tutoring process, when translated to code, constitutes the main function that controls the whole tutoring process by calling other functions. This recursive analysis of the tutoring steps ends when a composite tutoring process contains only atomic processes corresponding to simple statements.

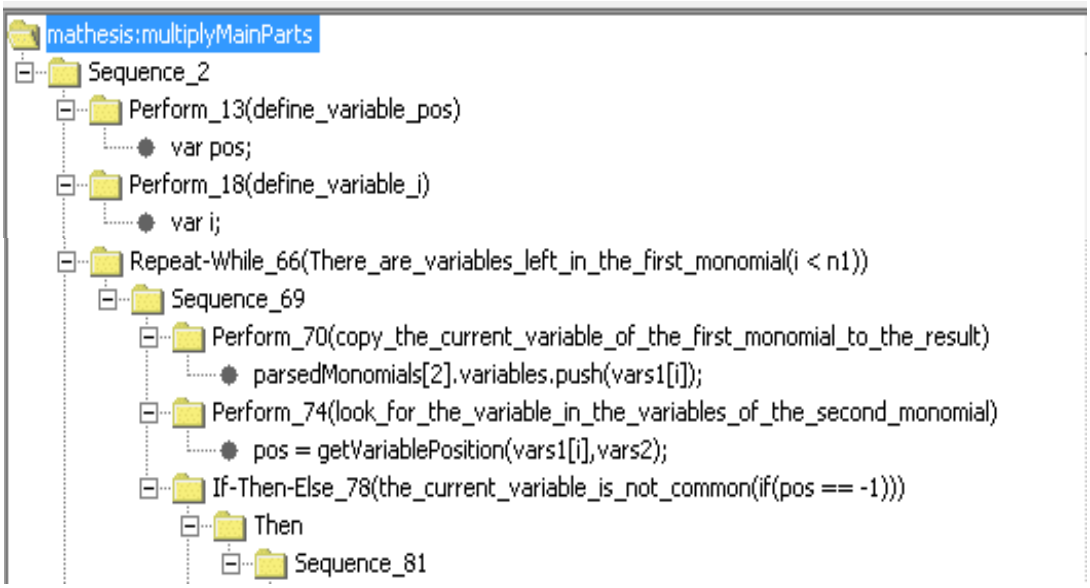


Fig. 9. Representation of the math domain task multiplyMainParts as a JavaScript function

For example, in the case of the execute-monomial-multiplication task, the execute-monomial-multiplication-Execution process (Figure 7c, Perform_13) is analysed in two other composite processes: multiplyCoefficients and multiplyMainParts. These two processes form the tutor's *mathematical domain expertise model*, which calculates the correct answer(s) in each step in order to be compared against the student's answer. Figure 9 shows part of the structure of process multiplyMainParts. Once again, there are two options for the meta-knowledge engineer on how to guide a domain expert in creating these processes:

1. The meta-knowledge engineer must develop the KE processes that, when executed, guide the domain expert in a step-by-step manner to implement them, or
2. the meta-knowledge engineer creates these tutoring (domain task) processes directly, using the Advanced KE Tools for Tutoring (Domain) Processes (Figure 7b), and then develops simpler KE processes that just guide the domain expert in selecting the former from the MATHESIS ontology.

The first option entails considerable workload for the meta-knowledge engineer but it is closer to the principles and objectives of the MATHESIS framework,. The second option is easier for the

meta-knowledge engineer but hides and obscures the KE expertise that was actually used to develop these domain task processes. To implement the first option, each JavaScript statement is represented by an instance of the JavaScriptStatement class, a subclass of AtomicProcess (Figure A2, Appendix). Following the OWL-S representational scheme, these instances are parameters to Perform constructs (Figure 9). 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, statement *pos = getVariablePosition (vars1[i], vars2)* (Figure 9, Perform_74) is an instance of class JavaScript_Assignment having three properties: hasAssignedVariable (value=*pos*), hasInvokedFunction (value= *getVariablePosition*) and hasArgumentsList (value = *(vars1[i], vars2)*) .

Such a detailed model of the JavaScript language allows the KE processes to guide the domain expert in building the tutor's (KBS) code by selecting the appropriate JavaScriptStatement subclass and the values of the related properties. For example, the assign_js_method process (Figure 10) creates JavaScript statements of the form `variable = object.method(arguments).`

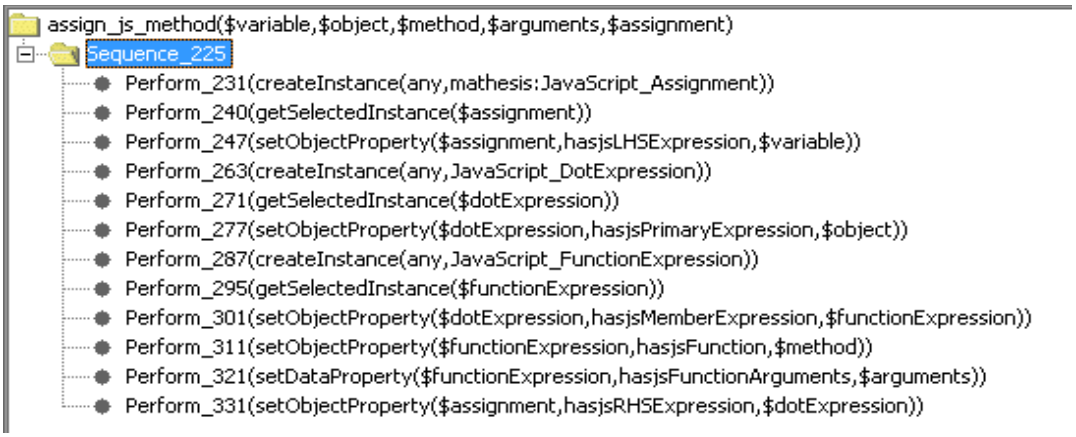


Fig. 10. The assign_js_method knowledge engineering process. This process creates JavaScript assignment statements of the form `variable = object.method(arguments).`

In the same time, from the values of these properties, the KE tools create and display the actual JavaScript code as shown in Figure 9 (the small disks under the Perform constructs). Therefore, a domain expert does not need to know the JavaScript syntax, but only needs to 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 domain expert. That is, the produced JavaScript code is syntactically correct, however whether this code exhibits the intended behaviour is a matter of correct design and analysis of the KE processes.

6.5 Interface model

The domain expert must also create within the MATHESIS ontology the user interface model. For web applications this can be a representation of the HTML Document Object Model (DOM). The tutor's placeholder for the interface model is kept by property hasTopInterfaceElement that holds the root element of the user interface, a Document instance (Figure 6, top right). We have developed an ontological representation of the HTML elements with their properties and values. When the domain expert creates the tutor's instance for the first time, the Tutor Initialization KE Tools automatically create the ontological representation of an empty HTML page. The representation of the HTML code and the corresponding DOM of the user interface for the monomial multiplication tutor are shown in Figure 11.

Each object defined in the HTML code is represented as an instance of the corresponding HTMLObject subclass (Document, Html, Head, Body, Applet, Web_Eq_Input_Control). Each HTMLObject instance has the corresponding HTML properties, like the *id* property, represented by HTMLProperty instances. In Figure 11, HTMLObject instances Web_Eq_Input_Control-1 and Web-Eq-

Input-Control-2 have their id HTML properties represented by HTMLProperty instances Web-Eq-Input-Control-1-id and Web-Eq-Input-Control-2-id correspondingly, pointed by object property hasHTMLProperty (Figure 11, bottom). The HTML values of these properties are represented by their corresponding ontological properties (*html-property-value* = "expressionInputControl").

The DOM tree structure is represented via two properties, hasFirstChild and hasNextSibling. This representation allows for bi-directional creation of the HTML part of the user interface:

1. The domain expert, either guided by the KE processes or using the Tutor Tools, creates within the MATHESIS ontology the representation of the DOM tree and then, by traversing the ontology, the translation tools generate the corresponding HTML code (top-down), or
2. the user interface is created using any Web-page authoring program and then 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 editing by the KE tools (bottom-up).

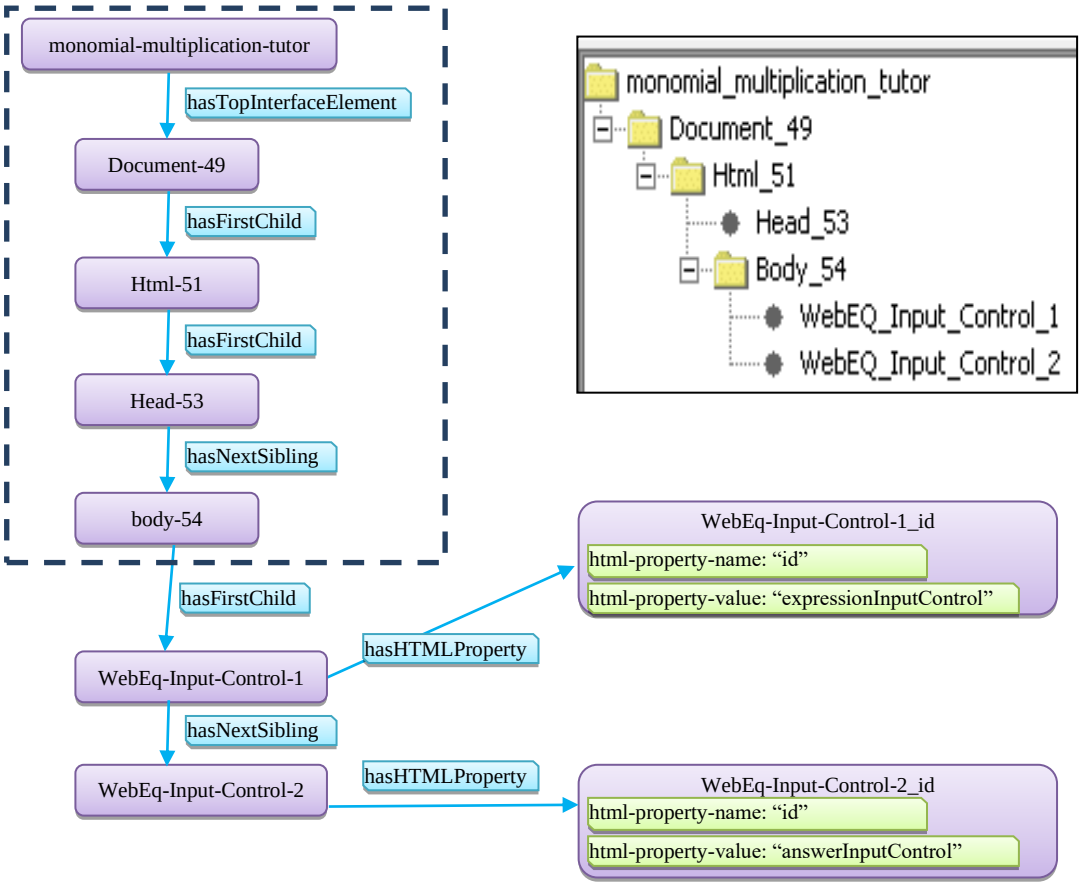


Fig. 11. The HTML User Interface DOM Ontological (left) and Visual (right, top) Representation

For different interface implementations like Java or Flash, appropriate ontological representations and translation programs must be developed.

The instances surrounded by the dotted line (Document_49, Html_51, Head_53 and Body_54) were created automatically by the Tutor Initialization Tools, while the rest were created by the execution of KE processes. Instances WebEq-Input-Control-1 and WebEq-Input-Control-2 were created by the domain expert. The first one is the interface element that is used to present the two monomials to be multiplied and represents the algebraic expression area of the MATHESIS Algebra Tutor (Figure 1b). The second one is used for the student answering area of the MATHESIS Algebra Tutor (Figure 1c). The meta-knowledge engineer associated monomial instances with the WebEq Input Control interface elements by setting object property has-Interface-

Elements of the generic monomial instance to WebEQ-Input-Control. Then, the meta-knowledge engineer created two KE process called define-interface-elements-for-input-knowledge-components and define-interface-elements-for-output-knowledge-components correspondingly that guide the domain expert in creating the two WebEq instances and naming them "expressionInputControl" and "answerInputControl" correspondingly (Figure 11, bottom right).

7. Knowledge reuse and scalability

Building from scratch any kind of knowledge-based system, like a model-tracing tutor, requires a tremendous effort, even with the use of authoring tools (Murray, 2003b). The main reason for this problem is the knowledge acquisition bottleneck. Domain experts are not trained in articulating their (teaching) expertise in a more abstract way. Highly trained knowledge engineers are needed to perform the following knowledge engineering tasks: Develop ontologies, represent the curriculum, represent teaching strategies and diagnostic procedures and create student models.

Existing authoring tools for knowledge engineering fall in either side of the breadth vs. depth trade-off: a) Tools that build tutors for very specific domains but with a deep domain model, or b) all-purpose authoring shells that build tutors for various domains but with shallow domain knowledge models. In addition to this trade-off, the problem of knowledge reuse - both for the developed tutoring (domain) knowledge as well as for the knowledge engineering (authoring) expertise – is severely restraining the widespread use of model-tracing tutors in particular and knowledge based systems in general in the real world.

As a solution to these problems, a three tiered meta-authoring framework was proposed by Murray (2003b). At its base there is a generic ITS authoring tool that requires the top level of authoring expertise like knowledge/ontological engineering, cognitive task analysis, instructional design, learning theories. This system is used to develop the middle tier, that is, reusable libraries of domain ontologies, student modelling rules, interface templates and generic teaching strategies. At the top level there are fairly simple tools for trained teachers that make use of the reusable libraries to develop powerful tutors for real world use. It is this meta-authoring model that the MATHESIS framework implements:

a) The base level is the ONTOMATH language and the Meta-Knowledge Engineering tools used by meta-knowledge engineers to represent knowledge engineering expertise as an ontology of executable knowledge engineering processes. These processes can guide domain experts in performing any kind of knowledge engineering tasks.
b) The middle tier corresponds to the MATHESIS Ontology, consisting of reusable parts of the tutors' cognitive, teaching and interface models. In the case of the monomial multiplication tutor development presented in section 6, these reusable parts where generic instances like monomial (domain concept), the ModelTracingAlgorithm (tutoring model), WebEq-Input-Control (HTML element) as well as reusable KE processes like the assign_js_method (JavaScript programming process). These generic instances are used by either the KE tools or the KE processes to create specific instances that represent the various parts of the monomial multiplication tutor.
c) The top level comprises the domain-specific knowledge engineering (authoring) tools, model-tracing in our case. The key characteristic of these tools is that they facilitate trained teachers (domain experts) to browse, locate and (re-)use the components of the middle tier. In the case of the MATHESIS model-tracing tutor authoring tools (Figure 7), this facilitation depends on the depth of the knowledge engineering ONTOMATH model, as it is the knowledge engineering processes that guide domain experts in locating and reusing existing or creating new tutors' parts. In that sense, each knowledge engineering process is a KE tool at a different level. This multi-level classification of knowledge engineering processes as knowledge engineering tools is illustrated even better by the fact that each one of the tools for tutor creation/selection, domain task creation/selection and teaching strategy model selection are shortcuts to common, top level knowledge engineering actions. They could be easily removed from the interface and replaced by knowledge engineering processes that would perform the same knowledge engineering

actions. The same holds for middle or low level knowledge engineering processes (tools). The more fine-grained they become the more knowledge engineering expertise they give to the system and the less expertise they demand from the domain expert.

We were faced with this problem of the KE model's granularity in an evaluation of the system. We asked a trained teacher of mathematics and expert computer user to use our tools to re-implement the monomial multiplication task of the MATHESIS Algebra Tutor. After 35 hours of training with the tools, he was capable of executing the knowledge engineering model and re-implement the monomial multiplication tutor. Then he tried to implement a monomial division tutor by executing the same model. We selected monomial division as it has minor differences from the multiplication task, mainly in the domain expertise model. When the author tried to implement tutoring process DivideMainParts that would perform the monomial division, the counterpart of tutoring process MultiplyMainParts (Figure 9), we realized that he had to modify the MultiplyMainParts process so that it would subtract the exponents of common variables ($x^m \div x^n = x^{m-n}$) rather than adding them ($x^m \cdot x^n = x^{m+n}$). In the MATHESIS framework there are two ways to solve this problem:

i. The non-expert author must use the Advanced Knowledge Engineering Tools for Tutoring Domain Processes (Figure 7b) to change the representation of the JavaScript statement that adds the exponents, so as to perform a subtraction instead of an addition. However, this elementary change entails a considerable raise at the expertise level required by the author; he must identify that statement (knowledge of JavaScript and programming) and change directly its ontological representation (knowledge of the MATHESIS ontology).
ii. The meta-knowledge engineer develops authoring processes that guide the domain expert in defining mathematical operations between mathematical objects. Then, the domain expert is guided by these processes to define whether the exponents (integers) of common variables are added (monomial multiplication), subtracted (monomial division) or even multiplied (monomial powers), like in $(x^m)^n = x^{m \cdot n}$.

Therefore, the MATHESIS framework deepens Murray's three tier structure of the tools to an arbitrary depth. Any time that the MATHESIS framework must cover the creation of tutors in a new domain, meta-knowledge engineers must create both the generic instances that represent declarative and procedural knowledge of the new domain as well as the KE processes that use them. Of course, parts of the new tutor(s) that are common with already developed tutors are readily reusable. In the case of the monomial division tutor monomial, ModelTracingAlgorithm and WebEq-Input-Control can be reused along with the KE processes that use them. This is possible due to the representation of the KE processes via the ONTOMATH language ontologically, that is in a declarative form, which makes them open for inspection and therefore mostly reusable (Gómez-Pérez, Fernández-López, Corcho, 2004). In existing knowledge engineering (authoring) systems, adding new KE knowledge would entail modifying the authoring program and adding new tools with their corresponding interfaces. This modification can be done only by the creators of the authoring program.

8. Discussion and further work

The MATHESIS meta-knowledge engineering framework primary goal is to spread the load of knowledge engineering over various levels of reusable knowledge engineering processes and over various knowledge engineers that can reuse them by browsing, locating and modifying them. For example, the ontological representation of any programming language like HTML or JavaScript and the knowledge engineering processes that create these representations can be standardized by authorised organizations and then used by any meta-knowledge engineer. Experts from any domain can develop libraries of ontological representations for their KBS and the knowledge engineering processes that create them. They could even develop their customized meta-knowledge engineering and domain-specific knowledge engineering tools and distribute them as Java applets. The implementation of the MATHESIS framework as an all-in-one package inside the Protégé OWL editor was done for easing the implementation of the system. The framework

could have been implemented with its parts distributed: The MATHESIS ontology being still an OWL ontology and the knowledge engineering tools being Java applications. Even Protégé's API used for grounding the ONTOMATH statements is a Java library. This distributed and collaborative knowledge engineering scheme can be supported by modern tools such as the WebProtégé collaborative ontology editor and knowledge acquisition tool (Tudorache, Nyulas, Noy, & Musen, 2013).

Of course, this multi-layered meta-knowledge engineering framework doesn't force the knowledge acquisition bottleneck and the efforts of knowledge engineering to disappear. The ontological representation of the tutor's (KBS) models, the structure of the knowledge engineering processes and the execution details that have to be taken into consideration for their development, suggest that there is no royal road to knowledge engineering. For the moment, the MATHESIS ontology contains the KE knowledge just for the development of two model tracing tutors, one for multiplication and one for division of monomials. Therefore, we do not claim that we have solved the problem of knowledge acquisition and reuse. Being in an experimental stage, we consider the MATHESIS framework as a *proof-of-concept* system. To investigate the issues of knowledge reuse and scalability, we plan to develop KE models for monomial by polynomial multiplication and polynomial multiplication. In the long term, we would like to develop a KE model that could create the entire MATHESIS Algebra Tutor. To facilitate this procedure, new knowledge engineering tools are needed, such as 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.

Acknowledgments

We thank the reviewers for their valuable comments. We also thank Mr. Sotiris Sakellaris, BSc and MSc in Mathematics, an expert mathematician and a power computer user, who offered to use our tools in the wild, by re-implementing the monomial multiplication and division tutors using the MATHESIS framework tools.

References

Aitken, J. S., & Sklavakis, D. (1999). Integrating problem solving methods into CYC. In T. Dean (Ed.) *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 627-633). San Francisco: Morgan Kaufman Publishers.

Aleven, V., McLaren, B., Sewall, J., & Koedinger, K. R. (2006). The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. *Proceedings of the 8th International Conference on Intelligent Tutoring Systems* (ITS 2006), (pp 61-70). Berlin: Springer.

Aleven, V., McLaren, B., Sewall, J., & Koedinger, K. R. (2009). A New Paradigm for Intelligent Tutoring Systems: Example-Tracing Tutors. *International Journal of Artificial Intelligence in Education,* Vol. 19 (pp.105-154).

Anderson, J.R. (1993). *Rules of the Mind*. Hillsdale, NJ: Erlbaum.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences*, Vol. 4 (2) (pp. 167-207).

Blessing., B.S., Gilbert, B. S., Ourada, S. & Ritter S. (2009). Authoring Model-Tracing Cognitive Tutors. *International Journal of Artificial Intelligence in Education,* Vol. 19 (pp.189-210)

Chandrasekaran, B. (1986). Generic Tasks for knowledge-based reasoning: the right level of abstraction for knowledge acquisition, *IEEE Expert*, Vol. 1 (pp. 23-30).

Clancey, W. J. (1985). Heuristic classification, *Artificial Intelligence* Vol. 27 (pp. 289-350).

Corbett, A. T. (2001). Cognitive Computer Tutors: Solving the Two-Sigma Problem. *Proceedings of the 8th International Conference on User Modeling 2001* (pp. 137-147). London: Springer.

Crowley, R.S., & Medvedeva, O. (2006). An intelligent tutoring system for visual classification problem solving. *Artificial Intelligence in Medicine*, Vol. 36 (pp. 85-117).

Denau, R., Dolbear, C., Hart, G., Dimitrova, V., & Cohn, A. (2011). Supporting Domain Experts to Construct Conceptual Ontologies: A Holistic Approach. *Web Semantics: Science, Services and Agents on the World Wide Web,*Vol. 9 (2) (pp. 113-127). Elsevier.

Dicheva, D., Mizoguchi, R., & Greer, J. (Eds.) (2009). *Semantic Web Technologies for e-learning, The Future of Learning,* Vol. 4. Amsterdam: IOS Press.

Gómez-Pérez, A., Fernández-López, M.,, & Corcho, O. (2004). Ontological Engineering: With Examples from the Areas of Knowledge Management, E-commerce and the Semantic Web. Berlin: Springer,

Hoffman, R. (1987). The Problem of Extracting the Knowledge of Experts From the Perspective of Experimental Psychology. *AI Magazine* (pp. 53-67).

Kaljurand, K. ACE View --- an ontology and rule editor based on Attempto Controlled English. *OWLED 2008*.

Kingston, J. (1995). Applying KADS to KADS: knowledge-based guidance for knowledge engineering. *Expert Systems*. Vol. 12(1), 15-26.

Koedinger, K. R., Anderson, J.R., Hadley, W.H., & Mark, M. A. (1997). Intelligent Tutoring Goes to School in the Big City, *International Journal of Artificial Intelligence in Education*, Vol. 8 (pp. 30-43).

Koedinger, K. R., & Corbett, A. (2006). Cognitive Tutors: Technology bringing learning science to the classroom. In K. Sawyer (Ed.) *The Cambridge Handbook of the Learning Sciences* (pp. 61-78), Cambridge University Press.

Lenat, D. B., & Guha, R. V. (1990). Building large Knowledge-based systems. Representation and inference in the Cyc project. Reading, Massachusetts: Addison-Wesley.

Lenat, D. B. (1995). CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38 (11).

Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., & Sycara, K. (2005). Bringing Semantics to Web Services: The OWL-S Approach, LNCS Vol. 3387 (pp. 26-42). Berlin: Springer.

Mellis, E., Andrès, J., Büdenbender, J., Frischauf, A., Goguadze, G., Libbrecht, P., Pollet, M. & Ullrich, C. (2001). A Generic and Adaptive Web-Based Learning Environment. *International Journal of Artificial Intelligence in Education,* 12, 385-407.

Mitrovic, A., Koedinger, K., & Martin, B. (2003). A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modelling. In P. Brusilovsky, A. Corbett & F. de Rosis (Eds.) *Proceedings of the 9th International Conference on User Modeling UM 2003* (pp. 313-322). LNAI 2702. Berlin: Springer-Verlag.

Mitrovich, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., & Holland, J. (2009). ASPIRE: An Authoring System and Deployment Environment for Constraint-Based Tutors. *International Journal of Artificial Intelligence in Education,* Vol. 19 (pp. 155-188)

Mizoguchi, R., Vankwelkenheusen, J., & Ikeda, M. (1995). Task Ontology for Reuse of Problem Solving Knowledge. *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing* (pp.46-59). Netherlands: IOS Press.

Mizoguchi, R., & Bourdeau, J. (2000). Using ontological engineering to overcome common AI-ED problems. *International Journal of Artificial Intelligence in Education*,Vol. 11 (pp. 107-121).

Mizoguchi, R. (2004). Tutorial on ontological engineering: part 3: Advanced course of ontological engineering. *New Generation Computing*. Vol. 22 (2) (pp. 198-220). Berlin: Springer.

Mizoguchi, R., Hayasi, Y., & Bourdeau, J. (2009). Inside a Theory-Aware Authoring System. In D. Dicheva, R. Mizoguchi & J. Greer (Eds.) *Semantic Web Technologies for e-learning: The Future of learning,* Vol. 4 (pp. 59-76). Amsterdam: IOS Press.

Murray, T. (2003a). An overview of intelligent tutoring system authoring tools: Updated Analysis of the State of the Art. In T. Murray, S. Ainsworth, & S. Blessing (Eds.) *Authoring Tools for Advanced Technology Learning Environments* (pp 491-544). Netherlands: Kluwer Academic Publishers.

Murray, T. (2003b). Principles for Pedagogy-Oriented Knowledge Based Tutor Authoring Systems. In T. Murray, S. Ainsworth, & S. Blessing (Eds.) *Authoring Tools for Advanced Technology Learning Environments* (pp 439-466). Netherlands: Kluwer Academic Publishers.

Paquette, L., Lebeau, J.-F., & Mayers, A. (2010). Authoring Problem-Solving Tutors: A Comparison Between CTAT and ASTUS. In Nkambou, R., Bourdeau, J., & Mizoguchi, R. (Eds.) *Advances in Intelligent Tutoring Systems* (pp 377-405). Heidelberg: Springer.

Puerta, A. R., & Musen, M. (1992). A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tasks. *Knowledge Acquisition,* Vol. 4 (pp. 171-196).

Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., Van de Velde, W., & Wielinga, B. (1999). *Knowledge Engineering and Manegement: The CommonKADS Methodology.* Cambridge MA: MIT Press.

Sklavakis, D. (1998). Implementing Problem Solving Methods in CYC. *MSc Dissertation*, Department of Artificial Intelligence, University of Edinburgh.

Sklavakis, D., & Refanidis, I. (2008). An Individualized Web-Based Algebra Tutor Based on Dynamic Deep Model-Tracing. *Proceedings of the Fifth Hellenic Conference on Artificial Intelligence* (SETN '08), (pp. 389-394). Heidelberg: Springer.

Sklavakis, D., & Refanidis, I. (2010). Ontology-Based Authoring of Intelligent Model-Tracing Math Tutors. *Proceedings of the Fourteenth International Conference on Artificial Intelligence* (AIMSA 2010), (pp. 201-210). Heidelberg: Springer.

Sklavakis, D., & Refanidis, I. (2013). MATHESIS: An Intelligent Web-Based Algebra Tutoring School. *International Journal of Artificial Intelligence in Education* Vol. 22 (2) (pp. 191-218). Amsterdam: IOS Press.

Suraweera, P., Mitrovic, A., Martin, B., Holland, J., Milik, N., Zakharov, K., & McGuigan, N. (2009). Using Ontologies to Author Constraint-Based Intelligent Tutoring Systems. In D. Dicheva, R. Mizoguchi & J. Greer (Eds.) *Semantic Web Technologies for e-learning: The Future of learning,* Vol. 4 (pp. 59-76). Amsterdam: IOS Press.

Tudorache, T., Nyulas, C., Noy, N.F., & Musen, M.A. (2013). WebProtégé: A Distributed Ontology Editor and Knowledge Acquisition Tool for the Web. *Semantic Web* Vol. 4 (1) (pp. 89-99). Amsterdam: IOS Press.

VanLehn, K., Jones, R. M., & Chi, M. T. H. (1992). A model of the self-explanation effect. *Journal of the Learning Sciences,* 2(1), 1-59.

VanLehn, K. (1999). Rule learning events in the acquisition of a complex skill: An evaluation of Cascade. *Journal of the Learning Sciences,* 8(2), 179-221.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., & Wintersgill, M. (2005). The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education* Vol. 15(pp. 147-204). Amsterdam: IOS Press.

VanLehn, K. (2006). The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education* Vol. 16 (3) (pp. 227-265). Amsterdam: IOS Press.

Wielinga, B.J., Schreiber, A.Th., & Breuker, J.A. (1992). KADS: A modeling approach to knowledge engineering. *Knowledge Acquisition* Vol. 4 (1) (pp. 5-53). Elsevier

Appendix

Table A1. The ONTOMATH Statements and their Operations

Browse Statements	**Purpose**
setSelectedClass*(className)*	Sets the Class named by *className* as selected in the Classes Panel
getSelectedClass*(className)*	Sets *className* to the name of the Class selected in the Classes Panel
setSelectedInstance*(instanceName)*	Sets the Instance named by *instanceName* as selected in the Instances Panel
getSelectedInstance*(instanceName)*	Sets *instanceName* to the name of the Instance selected in the Instances Panel
Collection Statements	**Purpose**
iteratorNext *(iterator, element)*	Sets *element* to the next element of *iterator*
String Statements	**Purpose**
strConcat*(newString,string1,string2)*	Concatenates *string1* and *string2* and returns the concatenated string to *newString*
Dialog Statements	**Purpose**
showMessageDialog*(message)*	Displays a Message Dialog with *message*
showInputDialog*(userInput, message, defaultValue)*	Displays an Input Dialog with *message* and *defaultValue*. Returns user input to *userInput*
Ontology_Editing Statements	**Purpose**
createInstance*(instanceName,className)*	Creates a new instance of class *className* named *instanceName*
copySelectedInstance*(newInstanceName)*	Creates a new copy of the instance that is currently selected in the Instances Panel with name *newInstanceName*.
createSubclass*(subclassName,superclassName)*	Creates a new subclass named *subclassName* of class *superclassName*
getObjectProperty*(instance,property,propertyValues)*	Gets the values of **object property** *property* of instance *instance* and stores them to variable *propertyValues.*
setObjectProperty*(instance,property,value)*	If **object** *property* of *instance* is **functional** (takes only one value), its value is set to *value*. Otherwise, *value* is added to the list of the property's values.
getDataProperty*(instance,property,propertyValues)*	Gets the values of **data property** *property* of instance *instance* and stores them to variable *propertyValues.*

setDataProperty*(instance,property,value)*	If **data** *property* of *instance* is **functional** (takes only one value), its value is set to *value*. Otherwise, *value* is added to the list of the property's values.
removePropertyValue*(instance,property,value)*	Removes instance *value* from the value(s) of property *property* of instance *instance.*
getLocalName*(instance, instanceName)*	Gets the local name of instance *instance* and stores it in variable *instanceName*.
setVariable(*variable,value)*	Sets the value of variable *variable* to *value.*

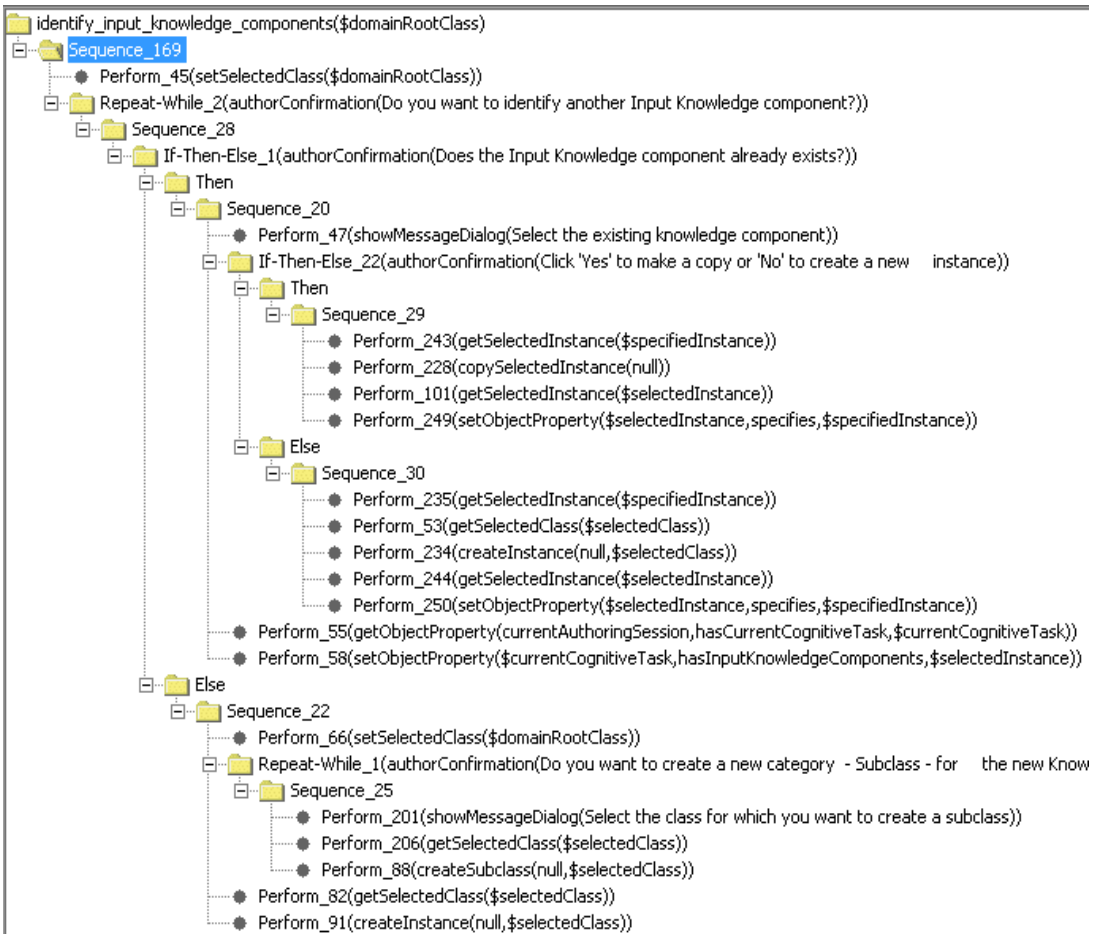


Fig. A1. The identify_input_knowledge_components knowledge engineering (ONTOMATH) process. By executing it a domain expert can create new instances of either an existing or a newly created monomial class

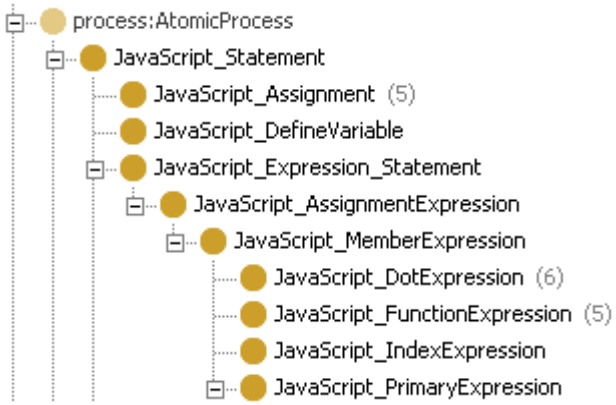


Fig.A2. Part of the JavaScript_Statement ontology