Ontology-Based Authoring of Intelligent Model-Tracing Math Tutors

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 Ontology, an OWL ontology developed within the MATHESIS project. The project aims at the development of an intelligent authoring environment for reusable model-tracing math tutors. 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. While the declarative knowledge is represented with the basic OWL components, i.e. classes, individuals and properties, the procedural knowledge 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. Based on such an ontological representation, a suite of authoring tools will be developed at the final stage of the project.

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

1 Introduction

Intelligent tutoring systems (ITSs) and especially model-tracing tutors have been proven quite successful in the area of mathematics [1]. Despite their efficiency, these tutors are expensive to build both in time and human resources. It is currently estimated that one hour of tutoring takes 200-300 hours of development [2]. Moreover, their development requires teams of Ph.D. level scientists in education, cognitive science and artificial intelligence.

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 [3].

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. What seems important in the case of ITSs is the creation of repositories of interoperable components that can be automatically retrieved and used by ITS shells or authoring tools [4].

This last research goal is in direct relationship with the research being conducted in the field of the Semantic Web and especially Semantic Web Services, a framework for the (semi-)automatic discovery, composition and invocation of web services and, in the case of ITSs, learning services [5].

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 but also for the authoring knowledge used by the tools. The basic research hypothesis of the project is that this goal can be better served by the basic Semantic Web technologies, that is by developing an OWL ontology for the description of the declarative knowledge and using the OWL-S web service description language [6] to represent the procedural knowledge for both the authoring tasks and the tutoring tasks of the developed tutors. This paper describes the MATHESIS ontology that implements this representational scheme.

The MATHESIS ontology is an OWL ontology developed with the Protégé-OWL ontology editor [7]. Its development is the second stage of the MATHESIS project. Aiming at the development of real-world, fully functional model-tracing math tutors, the project is being developed in a bottom-up approach. In the first stage the MATHESIS Algebra Tutor was developed in the domain of expanding and factoring algebraic expressions [8]. The tutor is web-based, using HTML and JavaScript. The authoring of the tutor as well as the code of the tutor were used to develop the MATHESIS ontology in an opportunistic way, that is in a bottom-up and top-down way, as it will be described later.

The rest of the paper is structured as follows: Section 2 gives a brief presentation of the process model of OWL-S which is the key technology of the MATHESIS ontology. Section 3 describes the part of the ontology that represents the model-tracing tutor(s), while Section 4 describes the representation of the authoring knowledge. Section 5 concludes with a discussion about the ontology and further work to complete the MATHESIS project.

2 The OWL-S Process Model

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

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

It is the last task that is of interest for the MATHESIS project and therefore we will focus on that. To support this task, OWL-S provides, among other things, a language for describing service compositions (see Fig.1).

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; composite processes correspond to actions that require multi-step protocols; finally, simple processes provide an abstraction mechanism to provide multiple views of the same process.

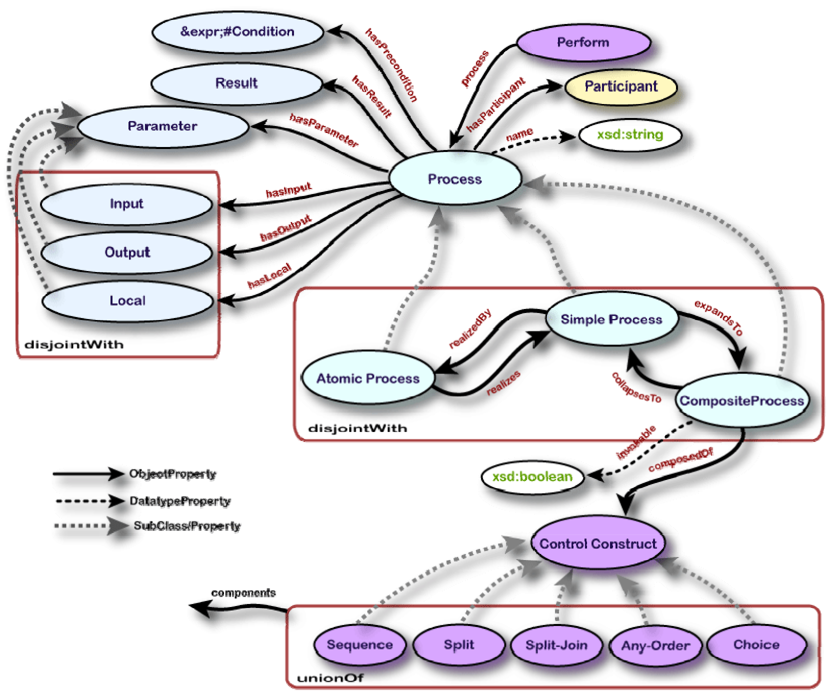


Fig. 1. Top level of OWL-S *process* ontology

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.

A CompositeProcess has a composedOf property by which is indicated the control structure of the composite process, using a ControlConstruct. Each ControlConstruct, in turn, is associated with an additional property called components to indicate the nested control constructs from which it is composed and, in some cases, their ordering. For example, any instance of the control construct Sequence has a components property that ranges over a ControlConstructList (a list of control constructs). Table 1 shows the most common control constructs that OWL-S supports.

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.

This tree-like representation of composite processes is the key characteristic of the OWL-S process model used in the MATHESIS ontology to represent both authoring and tutoring procedural knowledge, as it will be described in the subsequent sections.

Table 1. Common control constructs supported by 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.

3 Tutor Representation in 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. That means that during the authoring process and in the end, the result will be program code that implements the tutor. The ontology must be able to represent the program code. 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 [8].

The MATHESIS Algebra tutor is a Web-based, model-tracing tutor that teaches expanding and factoring of algebraic expressions: monomial and polynomial operations, identities, factoring. 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 that provides the same functionality as Equation Editor for Microsoft Word. Therefore, it is the representation of the HTML and JavaScript code that forms the low-level MATHESIS ontology of the tutor as described in the subsequent subsections.

3.1 Representation of the Tutor's HTML Code

The representation of the HTML code and the corresponding Document Object Model (DOM) of the user interface are shown in Figure 2. Each line of the HTML code is represented as an instance of the HTMLProgramLine class having three properties: the HTMLCode, hasNextLine and correspondingHTMLObject. The last one points to the HTMLObject defined by the code.

Each HTMLObject has the corresponding HTML properties as well as the hasFirst-Child and hasNextSibling which implement the DOM tree. Therefore, there are two representations of the HTML code enabling a bottom-up creation of the ontology (from HTML code to DOM) and a top-down (from the DOM to HTML code). Given

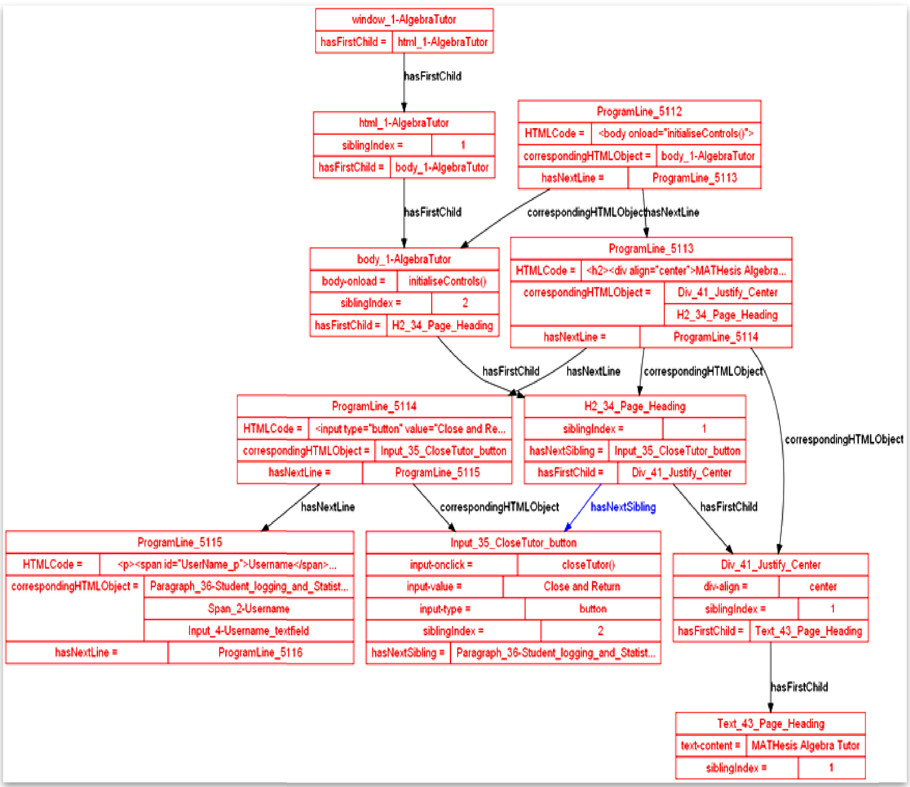

window_1-AlgebraTutor
hasFirstChild = html_1-AlgebraTutor
hasFirstChild
html_1-AlgebraTutor
siblingIndex = 1
hasFirstChild = body_1-AlgebraTutor
ProgramLine_5112
HTMLCode = <body onload="initialiseControls()">
correspondingHTMLObject = body_1-AlgebraTutor
hasNextLine = ProgramLine_5113
hasFirstChild
correspondingHTMLObject
hasNextLine
body_1-AlgebraTutor
body-onload = initialiseControls()
siblingIndex = 2
hasFirstChild = H2_34_Page_Heading
ProgramLine_5113
HTMLCode = <h2><div align="center">MATHesis Algebra...
correspondingHTMLObject = Div_41_Justify_Center
H2_34_Page_Heading
hasNextLine = ProgramLine_5114
hasFirstChild
hasNextLine
correspondingHTMLObject
ProgramLine_5114
HTMLCode = <input type="button" value="Close and Re...
correspondingHTMLObject = Input_35_CloseTutor_button
hasNextLine = ProgramLine_5115
H2_34_Page_Heading
siblingIndex = 1
hasNextSibling = Input_35_CloseTutor_button
hasFirstChild = Div_41_Justify_Center
correspondingHTMLObject
hasNextLine
correspondingHTMLObject
hasNextSibling
hasFirstChild
ProgramLine_5115
HTMLCode = <p><span id="UserName_p">Username</span>...
correspondingHTMLObject = Paragraph_36-Student_logging_and_Statist...
Span_2-Username
Input_4-Username_textfield
hasNextLine = ProgramLine_5116
Input_35_CloseTutor_button
input-onclick = closeTutor()
input-value = Close and Return
input-type = button
siblingIndex = 2
hasNextSibling = Paragraph_36-Student_logging_and_Statist...
Div_41_Justify_Center
div-align = center
siblingIndex = 1
hasFirstChild = Text_43_Page_Heading
hasFirstChild
Text_43_Page_Heading
text-content = MATHesis Algebra Tutor
siblingIndex = 1

Fig. 2. The HTML code and the corresponding DOM representation

the appropriate parsing tools, one can build the interface in any Web-page authoring program and then generate from the HTML code the DOM ontology or, in the reverse order, the tutor authoring tools will create the DOM representation and then the HTML code.

3.2 Representation of the Tutor's JavaScript Code

The representation of the JavaScript code is shown in Figure 3. Each line of the JavaScript code is represented as an instance of the JavaScript_ProgramLine class having three properties: the javascriptCode, hasNextLine and hasJavaScriptStatement. The last one points to a JavaScript_Statement instance which is an AtomicProcess of OWL-S process model.

Once again, there are two representations of the JavaScript code enabling a bottom-up creation of the ontology (from JavaScript code to JavaScript_Statement atomic processes) and a top-down (from the JavaScript_Statement atomic processes to JavaScript code). Given the appropriate parsing tools, one can take JavaScript and then generate the corresponding JavaScript_Statement atomic processes or, in the reverse order, the tutor authoring tools will create the JavaScript_Statement atomic processes and then the JavaScript code.

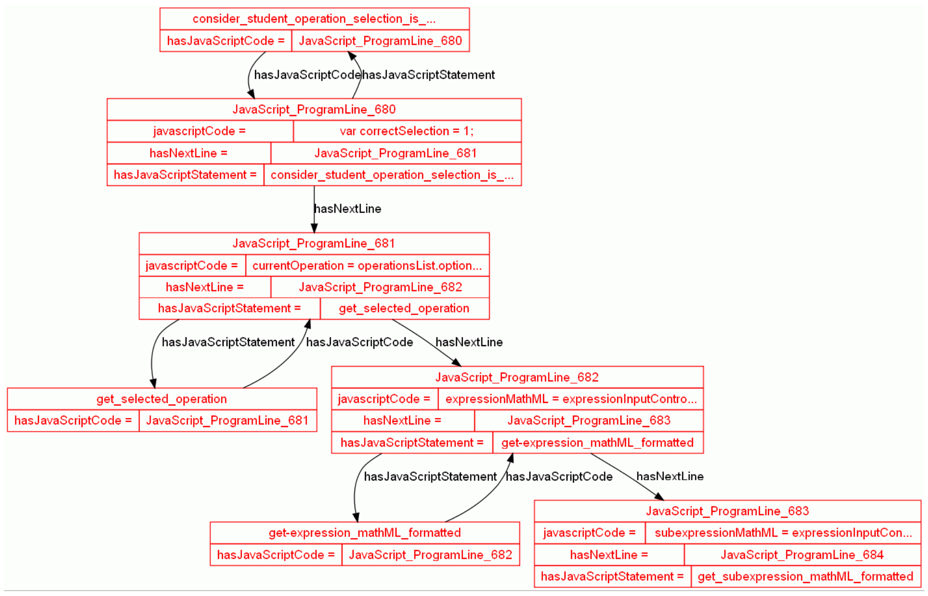
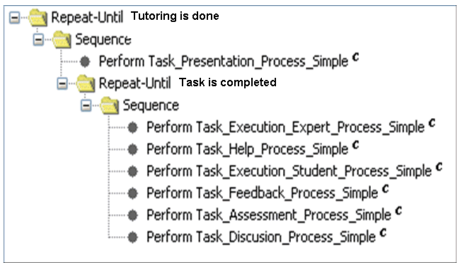

consider_student_operation_selection_is_...
hasJavaScriptCode = | JavaScript_ProgramLine_680
hasJavaScriptCode
hasJavaScriptStatement
JavaScript_ProgramLine_680
javascriptCode = | var correctSelection = 1;
hasNextLine = | JavaScript_ProgramLine_681
hasJavaScriptStatement = | consider_student_operation_selection_is_...
hasNextLine
JavaScript_ProgramLine_681
javascriptCode = | currentOperation = operationsList.option...
hasNextLine = | JavaScript_ProgramLine_682
hasJavaScriptStatement = | get_selected_operation
hasJavaScriptStatement
hasJavaScriptCode
hasNextLine
get_selected_operation
hasJavaScriptCode = | JavaScript_ProgramLine_681
JavaScript_ProgramLine_682
javascriptCode = | expressionMathML = expressionInputContro...
hasNextLine = | JavaScript_ProgramLine_683
hasJavaScriptStatement = | get-expression_mathML_formatted
hasJavaScriptStatement
hasJavaScriptCode
hasNextLine
get-expression_mathML_formatted
hasJavaScriptCode = | JavaScript_ProgramLine_682
JavaScript_ProgramLine_683
javascriptCode = | subexpressionMathML = expressionInputCon...
hasNextLine = | JavaScript_ProgramLine_684
hasJavaScriptStatement = | get_subexpression_mathML_formatted

Fig. 3. The JavaScript code and the corresponding JavaScript_Statement Atomic processes

3.3 Representation of the Tutoring Model

Model-tracing tutors are named after their fixed tutoring model, the model-tracing algorithm. Being procedural knowledge, the model-tracing algorithm is represented in the MATHESIS ontology as a composite process named Model_Tracing_Algorithm (Figure 4). Each step of the algorithm is also a composite process (notice the small 'c' letter at the end of the names). That means that each step can be further elaborated by another algorithm (composite process). For example the Task_Execution_Expert_Process step can be described by an algorithm (not developed yet) that performs other composite processes like Step-Execution-Expert-Process, Task-Asked-Identification-Expert-Process, Task-Given-Identification-Expert-Process, Task-Sub-Task-Execution-Expert-Process (Figure 5).


```
Repeat-Until  Tutoring is done
  Sequence
    Perform Task_Presentation_Process_Simple c
    Repeat-Until  Task is completed
      Sequence
        Perform Task_Execution_Expert_Process_Simple c
        Perform Task_Help_Process_Simple c
        Perform Task_Execution_Student_Process_Simple c
        Perform Task_Feedback_Process_Simple c
        Perform Task_Assessment_Process_Simple c
        Perform Task_Discusion_Process_Simple c
```

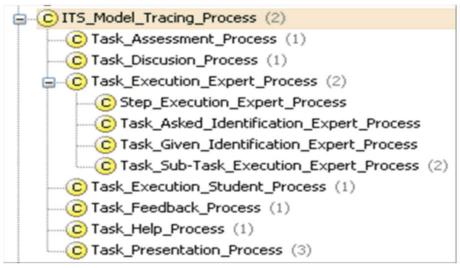
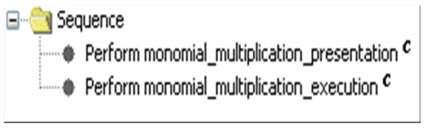

```
ITS_Model_Tracing_Process (2)
  Task_Assessment_Process (1)
  Task_Discusion_Process (1)
  Task_Execution_Expert_Process (2)
    Step_Execution_Expert_Process
    Task_Asked_Identification_Expert_Process
    Task_Given_Identification_Expert_Process
    Task_Sub-Task_Execution_Expert_Process (2)
  Task_Execution_Student_Process (1)
  Task_Feedback_Process (1)
  Task_Help_Process (1)
  Task_Presentation_Process (3)
```

Fig. 4. The Model-Tracing-Algorithm process **Fig. 5.** The model-tracing processes taxonomy

During the authoring of a specific tutor, the authoring tools will parse the tree of the Model-Tracing-Algorithm composite process and will invoke for each tutoring task a corresponding authoring task represented also as a composite process in order to implement the tutoring task for the specific tutor (see Section 4).

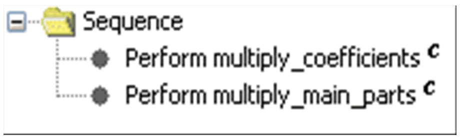
3.4 Representation of the Domain Expert Model

In a model-tracing tutor, the Domain Expert Model executes the next step of the problem and produces the correct solution(s) to compare them with the student's proposed solution. If the solution step is simple, then it is represented as an instance of the atomic process JavaScript_Statement (see Section 3.2). If the step is complex, then it is represented as a composite process. This analysis ends when the produced composite processes contain only atomic processes, that is JavaScript_Statement instances. An example for the monomial multiplication task from the MATHESIS Algebra tutor will



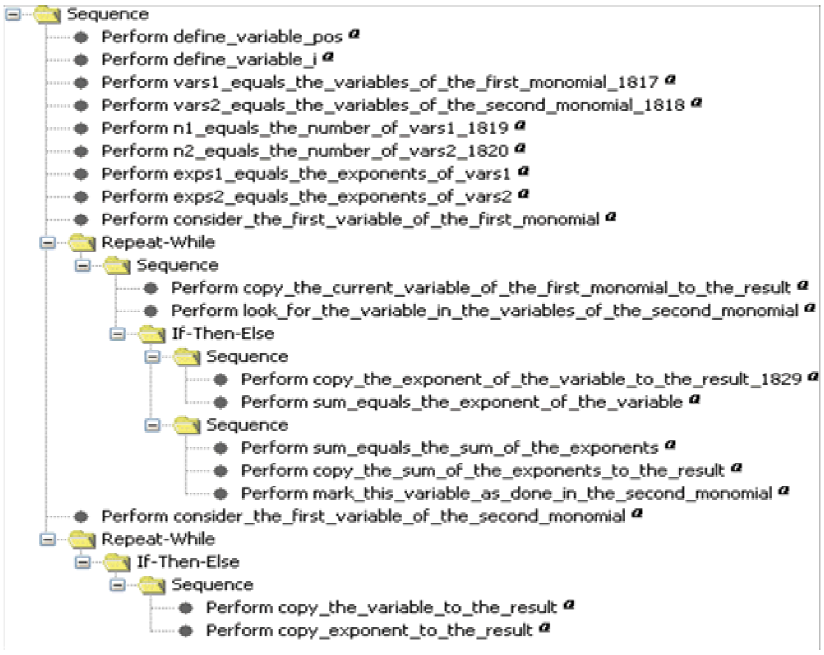
```
Sequence
    Perform monomial_multiplication_presentation c
    Perform monomial_multiplication_execution c
```

Fig. 6. Part of the monomial-multiplication model-tracing algorithm composite process



```
Sequence
    Perform multiply_coefficients c
    Perform multiply_main_parts c
```

Fig. 7. The monomial-multiplication-execution composite process



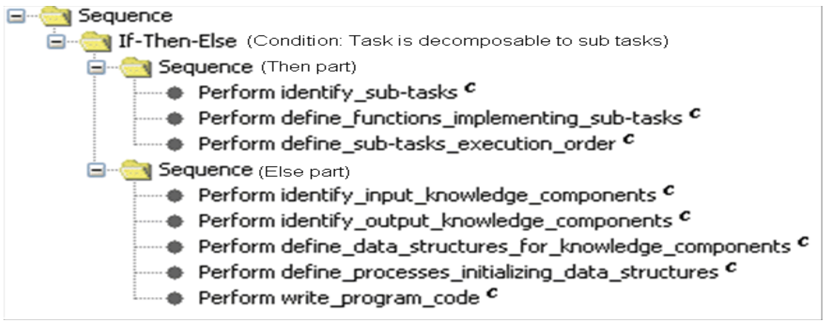
```
Sequence
    Perform define_variable_pos a
    Perform define_variable_j a
    Perform vars1_equals_the_variables_of_the_first_monomial_1817 a
    Perform vars2_equals_the_variables_of_the_second_monomial_1818 a
    Perform n1_equals_the_number_of_vars1_1819 a
    Perform n2_equals_the_number_of_vars2_1820 a
    Perform exps1_equals_the_exponents_of_vars1 a
    Perform exps2_equals_the_exponents_of_vars2 a
    Perform consider_the_first_variable_of_the_first_monomial a
    Repeat-While
        Sequence
            Perform copy_the_current_variable_of_the_first_monomial_to_the_result a
            Perform look_for_the_variable_in_the_variables_of_the_second_monomial a
            If-Then-Else
                Sequence
                    Perform copy_the_exponent_of_the_variable_to_the_result_1829 a
                    Perform sum_equals_the_exponent_of_the_variable a
                Sequence
                    Perform sum_equals_the_sum_of_the_exponents a
                    Perform copy_the_sum_of_the_exponents_to_the_result a
                    Perform mark_this_variable_as_done_in_the_second_monomial a
    Perform consider_the_first_variable_of_the_second_monomial a
    Repeat-While
        If-Then-Else
            Sequence
                Perform copy_the_variable_to_the_result a
                Perform copy_exponent_to_the_result a
```

Fig. 8. The multiply-main-parts composite process

illustrate this technique. In Figure 6 a part of the model-tracing algorithm is presented as implemented for the monomial multiplication task according to Section 3.3. As the monomial_multiplication_execution step is a complex one, it is represented as a composite process with two other composite processes, multiply_coefficients and multiply_main_parts (Figure 7). Finally, in Figure 8 the multiply_main_parts process is described, containing only instances of the atomic process JavaScript_Statement (see Section 3.2).

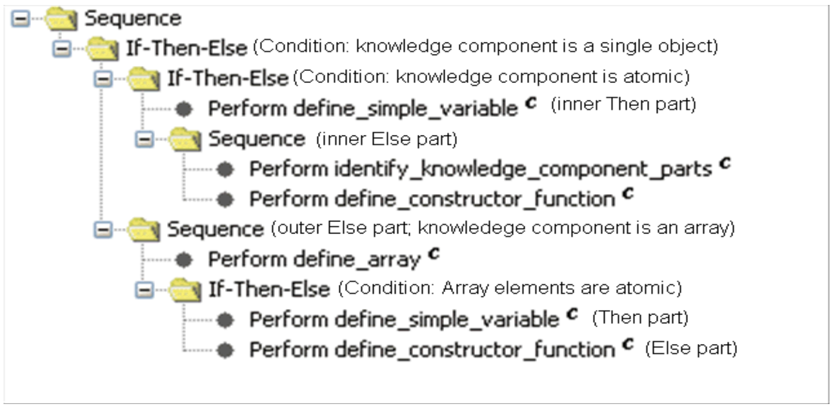
4 Authoring Knowledge Representation in MATHESIS Ontology

As mentioned in Section 3.3, for each tutoring task of the model-tracing algorithm, there is a corresponding authoring task in the MATHESIS ontology, represented also as a composite process. The authoring_task_execute_task_by_expert (Figure 9) for example corresponds to the Task_Execution_Expert_Process_Simple tutoring task (see Figure 4). One of the composite processes that form this authoring task, the define_data_structures_for_knowledge_components is shown in Figure 10.



```
Sequence
  If-Then-Else (Condition: Task is decomposable to sub tasks)
    Sequence (Then part)
      Perform identify_sub-tasks c
      Perform define_functions_implementing_sub-tasks c
      Perform define_sub-tasks_execution_order c
    Sequence (Else part)
      Perform identify_input_knowledge_components c
      Perform identify_output_knowledge_components c
      Perform define_data_structures_for_knowledge_components c
      Perform define_processes_initializing_data_structures c
      Perform write_program_code c
```

Fig. 9. The authoring process for the Execute-Task-By-Expert tutoring task



```
Sequence
  If-Then-Else (Condition: knowledge component is a single object)
    If-Then-Else (Condition: knowledge component is atomic)
      Perform define_simple_variable c (inner Then part)
      Sequence (inner Else part)
        Perform identify_knowledge_component_parts c
        Perform define_constructor_function c
    Sequence (outer Else part; knowledege component is an array)
      Perform define_array c
      If-Then-Else (Condition: Array elements are atomic)
        Perform define_simple_variable c (Then part)
        Perform define_constructor_function c (Else part)
```

Fig. 10. The define-data-structures authoring process

Based on all the above representations, the overall authoring process will have as follows: The tools will parse the model-tracing algorithm (Section 3.3). For each step of the algorithm, the corresponding authoring process will be called and traced. This authoring process will guide the author in creating recursively the various parts of the tutor (Sections 3.1, 3.2, 3.4); as a consequence, any newly created tutor part becomes new knowledge in the ontology to be used later. In addition, the authoring process can present known issues, design principles, examples and existing implementations for the tutoring task to which it corresponds, maximizing that way knowledge re-use.

5 Discussion and Further Work

In an overview of intelligent tutoring system authoring tools [3], it is suggested that authoring tools should support:

- *Interoperability*: Tutors should be able to work with other systems providing *inspect-ability* (provide information to other applications), *record-ability* (broadcast data about their state) and *script-ability* (controlled by other applications)
- *Re-usability*: Tutors should be able to be used in a context different from their original one
- *Durability* and *Scalability*: Tutors should be able to cover big-sized domains and accommodate many simultaneous users
- *Accessibility*: Learner or content developer should locate and download material in a reasonable time.

Even in this preliminary form, the MATHESIS ontology provides a proof-of-concept that it can serve as the basis for the development of authoring tools and implemented tutors that will match these criteria. The main reason for this claim is that the ontology provides an open, modular and multi-level representation (ranging from conceptual design to program code) of both authoring and tutoring knowledge. In addition, being an OWL ontology with all procedural knowledge encoded as OWL-S processes opens a window towards semantic web services and Service Oriented Architectures [9]. Finally, the tutors, implemented in HTML and JavaScript, possess in a great extent the listed characteristics.

Of course, it is obvious that a lot of work has to be done: the ontology must be extended, refined and in a great extend formalized. This will be done by representing the whole MATHESIS Algebra tutor into the ontology. Because of the tremendous workload this task entails, an initial set of authoring tools are needed: parsers for HTML to/from MATHESIS interface model and for JavaScript to/from MATHESIS JavaScript_Statements/Program code translation; interpreters for the authoring and tutoring OWL-S processes; visualization tools for the authoring processes, the tutoring model (model-tracing algorithm) and the tutor parts being developed. 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. Aleven, V., McLaren, B., Sewall, J., Koedinger, K.R.: The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. In: Ikeda, M., Ashley, K.D., Chan, T.-W. (eds.) ITS 2006. LNCS, vol. 4053, pp. 61–70. Springer, Heidelberg (2006)
3. Murray, T.: An Overview of Intelligent Tutoring System Authoring Tools: Updated Analysis of the State of the Art. In: Authoring Tools for Advanced Technology Learning Environments, pp. 491–544. Kluwer Academic Publishers, Netherlands (2003)
4. Davies, J., Studer, R., Waren, P. (eds.): Semantic Web Technologies: trends and research in ontology-based systems. John Wiley & Sons Ltd., England (2006)
5. Dicheva, D., Mizogichi, R., Capuano, N., Harrer, A. (eds.): Proceedings of the Fifth International Workshop on Ontologies and Semantic Web for E-Learning, SWEL 2007 (2007), `http://compsci.wssu.edu/iis/Papers/SWEL07-Proceedings.pdf`
6. Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing Semantics to Web Services: The OWL-S Approach. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 26–42. Springer, Heidelberg (2005)
7. The Protégé ontology editor, `http://protege.stanford.edu/overview/protege-owl.html`
8. 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)
9. Gutiérrez-Carreón, G., Daradoumis, T., Jorba, J.: Exploring Semantic Description and Matching Technologies for Enhancing the Automatic Composition of Grid-based Learning Services. In: Proceedings of the Fifth International Workshop on Ontologies and Semantic Web for E-Learning, SWEL 2007 (2007), `http://compsci.wssu.edu/iis/Papers/SWEL07-Proceedings.pdf`