Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2009), *23*, 53–69. Printed in the USA.
Copyright © 2009 Cambridge University Press 0890-0604/09 $25.00
doi:10.1017/S0890060409000109

A dynamic knowledge modeler

ROBERT HARRISON AND CHRISTINE W. CHAN

Energy Informatics Laboratory, Faculty of Engineering, University of Regina, Regina, Saskatchewan, Canada

(RECEIVED September 22, 2007; ACCEPTED June 24, 2008)

Abstract

This paper presents the development and application of a software tool for modeling knowledge to be used in knowledge-based systems or the Semantic Web. The inferential modeling technique, which is a technique for modeling the static and dynamic knowledge elements of a problem domain, provided the basis for the tool. A survey of existing knowledge modeling tools revealed they typically failed to provide support in four main areas: support for an ontological engineering methodology or technique, support for dynamic knowledge modeling, support for dynamic knowledge testing, and support for ontology management. Dyna, a Protégé plug-in, has been developed, which supports the Inferential Modeling Technique, dynamic knowledge modeling, and dynamic knowledge testing. Protégé and Dyna are applied for constructing an ontological model in the domain of selecting a remediation technology for petroleum contaminated sites. Dynamic knowledge testing in Dyna enabled creation of a more complete knowledge model.

Keywords: Knowledge Engineering; Ontology; Semantic Web; Software Engineering

1. INTRODUCTION

Knowledge-based systems (KBS) are often limited to problem solving in a narrow domain of knowledge because of high development costs. The major cost is construction of the knowledge base. Knowledge bases are often constructed from scratch because KBS development occurs in a distributed, heterogeneous environment consisting of different locations, system types, knowledge representations, and so forth, making it very difficult to share and reuse existing knowledge base components.

Sharing and reusing knowledge can help reduce costs and make it possible to create systems capable of more powerful problem solving. The Semantic Web (SW) is the next evolutionary step for the Web. The SW aims to provide semantics to data on the Web, enabling computers to more easily share and perform problem solving on the data (Berners-Lee et al., 2001). SW technology can be used to share and reuse knowledge between KBSs in a distributed, heterogeneous environment.

A requirement for the SW is that data on the Web are structured semantically for machine processing, and ontologies can contribute to that objective. Ontologies can also become the basis for building KBSs. An ontology is an "explicit specification of a shared conceptualization" (Gruber, 1993); it can be used for structuring the knowledge in a KBS and on the SW. The main benefit of an ontology is that it enables the sharing and reuse of application domain knowledge across distributed and heterogeneous software systems (Guarino, 1998). Ontologies implemented in XML-based languages, such as Resource Description Framework (RDF; http://www.w3.org/RDF/) and Web Ontology Language (OWL; http://www.w3.org/TR/owl-features/), enable different KBS development groups or different SW applications to share and reuse their knowledge and data. However, the construction of ontologies is time consuming and costly.

Software tools can help reduce the effort required to construct ontologies and knowledge bases. The general objective of our work is to provide software tool support for knowledge creation for the SW. Our approach involves examining existing software tools for knowledge creation, and this examination provided the basis for the design of a new tool for supporting knowledge creation. The new tool aims to address deficiencies noted in some existing tools; it was applied for building an application ontology in the petroleum contamination remediation selection domain.

This paper is organized as follows. Section 2 presents relevant background literature. Section 3 describes the development of a tool for dynamic knowledge modeling. Section 5 discusses the tool for developing an application ontology. Section 6 provides some conclusions and Section 7 discusses directions for future work.

Reprint requests to: Robert Harrison, Energy Informatics Laboratory, Faculty of Engineering, University of Regina, Regina, Saskatchewan S4S 0A2, Canada. E-mail: harrisor@uregina.ca

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

2. BACKGROUND LITERATURE

2.1. Inferential modeling technique (IMT)

The IMT is a technique for modeling the static and dynamic knowledge elements of a problem domain. Static knowledge consists of observable domain objects (classes), attributes of classes, and relationships between classes. Dynamic knowledge consists of tasks (or processes) that manipulate the static knowledge to achieve an objective. The IMT is an iterative process of knowledge modeling, and the procedure is listed below. For further details on the technique, see Chan (2004).

1. Specify static knowledge
   a. Specify the physical objects in the domain
   b. Specify the properties of objects
   c. Specify the values of the properties or define the properties as functions or equations
   d. Specify the relations associated with objects and properties as functions or equations
   e. Specify the partial order of the relations in terms of strength factors and criteria associated with the relations
   f. Specify the inference relations derived from the objects and properties
   g. Specify the partial order of the inference relations in terms of strength factors and criteria associated with the relations
2. Specify the dynamic knowledge
   a. Specify the tasks
   b. Decompose the tasks identified into inference structures or subtasks
   c. Specify the partial order of the inference and subtask structures in terms of strength factors and criteria
   d. Specify strategic knowledge in the domain
   e. Specify how strategic knowledge identified is related to task and inference structures specified
3. Return to Step 1 until the specification of knowledge types is satisfactory to both the expert and knowledge engineer.

2.2. Knowledge modeling system

Knowledge Modeling System (KMS) is a software tool for modeling static and dynamic knowledge as defined in the IMT. KMS contains a "class" module for modeling static knowledge and a "task" module for modeling dynamic knowledge. The class module enables the user to model concepts or classes and properties of classes. The class module also supports the creation of inheritance and association relationships between classes. The task module enables the user to create tasks and objectives. A type of strategic knowledge can be specified by adding a prioritized list of tasks to an objective. The task module also supports the user in linking static knowledge created in the class module to tasks. Task behavior or operations that manipulate objects can also be defined. KMS provided a basis for the work presented in this paper.

2.3. Ontology construction tools

Software tools enable ontology authors to ignore the complexities of the ontology languages used in developing application ontologies. The tools support efficient application development, so that the development process can be completed with fewer errors and involves a lower learning curve. From a survey of research work on ontology tools, we found four areas that require improvement (Harrison & Chan, 2005), which are discussed in the following subsections.

2.3.1. *Ontological engineering methodology*

The process of developing an ontology consists of activities in three different areas. First, there are ontology development activities such as specification, implementation, and maintenance. Second, there are ontology management activities such as reusing existing ontologies and quality control. Third, there are ontology support activities such as knowledge acquisition and documentation (Gomez-Perez et al., 2005). An ontological engineering methodology specifies the relationships among the activities and when the activities should be performed.

A tool that supports an ontological engineering methodology or technique can expedite the ontological engineering process. Currently, there are few tools that directly support an ontological engineering methodology; some existing tools include OntoEdit, which supports On-To-Knowledge (Fensel et al., 2002); WebODE, which supports METHONTOLOGY (Gomez-Perez et al., 2003); and KMS, which supports IMT (Chan, 2004).

On-To-Knowledge is an iterative methodology that consists of five steps: feasibility study, kickoff, refinement, evaluation, and maintenance (Gomez-Perezet et al., 2005). OntoEdit provides support for ontology implementation, which happens in the third step of refinement, and maintenance (Fensel et al., 2002).

METHONTOLOGY is based on the software development process and provides support for the entire process during the development life cycle of an ontology. WebODE's ontology editor consists of a number of services that enable it to support various activities in the ontology life cycle defined in METHONTOLOGY. WebODE's services include ontology implementation, documentation, reasoning, and evaluation (Gomez-Perez et al., 2005).

As discussed in Section 2.1, the IMT is a technique for developing a classification of the knowledge elements of a domain. KMS provides automated support for developing the static and dynamic knowledge elements of a domain, and the tool was developed based on the IMT (Chan, 2004).

2.3.2. *Ontology modeling*

An ontological engineering tool enables developing a knowledge or ontological model of a problem domain. A brief survey

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

of the different ontological engineering support tools reveal there are diverse methods employed. For example, the method used by Protégé (Gennari et al., 2003), Ontolingua (McGuiness et al., 2003), and OntoBuilder (Gal et al., 2006) for modeling ontologies involves listing all the concepts in a hierarchical tree, and input fields are provided to capture characteristics of concepts. A graphical method of modeling ontologies is employed in tools such as KAON OI-Modeler (Gabel et al., 2004) and Protégé OWLViz Plug-in (Horridge, 2005*b*), in which the representational method uses nodes to represent concepts and edges to represent relationships between concepts.

The Unified Modeling Language (UML) tools are commonly used for visual representation and communication of software design. Because of the similarities between ontologies and object-oriented design, UML class diagrams can be used for modeling ontology classes and their properties, and relationships between classes (Cranefield et al., 2000). However, UML's expressiveness for modeling ontologies is limited; for example, standard UML cannot express more advanced ontologies that contain descriptive logic (Gasevic et al., 2005).

Because of the limitations of UML, there is ongoing research work that aims to develop a graphical notation for ontologies, called the Ontology Definition Metamodel (ODM; Gasevic et al., 2005).

Existing ontological engineering tools can model static knowledge to varying degrees. One issue is data type support. Not all tools are capable of supporting all data types (string, integer, float, etc.). The data types supported by the tool should be investigated before the modeling process begins.

Although most tools can support representation of static knowledge, there are significant difficulties when these tools are used for modeling dynamic knowledge. The general procedure to model dynamic knowledge in tools such as Protégé, KAON OI-Modeler, and KMS consists of the following three steps:

1. Create classes to represent task and objective. These are the main structures that will be used for modeling dynamic domain knowledge. It should be noted that this step is not required in KMS, as it was designed for modeling dynamic domain knowledge and most of the necessary structures are built in.
2. Model the static domain knowledge.
3. Model the dynamic domain knowledge by instantiating the task and objective classes that were created in Step 1.

To better illustrate the problems encountered when modeling dynamic knowledge in existing tools, the above steps using Protégé are applied to model a small part of the petroleum contamination remediation selection ontology. Protégé uses a form-based user interface for inputting knowledge; there are windows containing text fields for inputting knowledge. The application of the above steps using Protégé is described as follows:

1. Create classes to represent task and objective.
   a. Create a new Protégé project for containing the task and objective classes.
   b. Create a class for representing task knowledge with the following characteristics:

      Class Name: Task
      Super Class: owl:Thing
      Properties:
      — name (single string)
      — behavior (single string)
      — inputs
      — objects
      — output (single)
      — subtasks (multiple Tasks)

      The **Task** class properties include **behavior**, associated **objects**, and a list of **sub** (or dependent) tasks. In task interactions, data can be exchanged. To capture this important information, two additional properties, **inputs** and **output** are added to the **Task** class. A tool that supports inputs and outputs facilitates the ontology author in considering the detailed interactions between tasks and objectives; hence, a more complete model is likely to be generated.
   c. Task priority is a type of strategic knowledge that is most likely to be found in an industrial domain, such as petroleum contamination remediation selection. For any particular objective, the task priority specifies which tasks are required and the order in which they are to be performed. Task priority can be embedded in a prioritized list of tasks inside the objective class. However, the existing ontology tools, including Protégé, lack support for specifying the order (or priority) of specific items. This problem can be solved with the addition of an intermediate class, called **TaskPriority**, to relate a task to a priority. The class for representing **TaskPriority** is shown as follows:

      Class Name: TaskPriority
      Super Class: owl:Thing
      Properties:
      — priority (single int)
      — task (single Task)

      The **TaskPriority** class contains the properties **task** (an instance of a Task) and **priority** (an integer), thus enabling the association of a priority to a task.
   d. Create a class to represent an objective with the following characteristics:

      Class Name: Objective
      Super Class: owl:Thing
      Properties:
      — taskPriorityList (multiple TaskPriority)

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

The **Objective** class has a property for the **taskPriorityList**.

2. Model the static domain knowledge.
   a. Create a new Protégé project for containing the static domain knowledge.
   b. Create classes, properties, and relations.
3. Model the dynamic domain knowledge.
   a. Create a new Protégé project for containing the dynamic domain knowledge.
   b. Import both the Protégé project created in Step 1 (task, objective, and TaskPriority classes) and the Protégé project created in Step 2 (static domain knowledge) in the dynamic domain knowledge project.
   c. Choose an objective to model. In this example, the objective, Determine Contamination Level would be modeled. This is done by creating an instance of the objective class identified as **DetermineContaminationLevel**. The specification of task priority (strategic knowledge) is discussed later.
   d. This step describes how to create a task that is required to achieve the chosen objective. In this example, a task to Calculate Weighted Normalized Benzene Concentration will be created. To model this task, the Task class is instantiated and identified with the name **CalculateWeightedNormalizedBenzeneConcentration**. Next the task behavior and objects can be modeled. These characteristics are shown as follows:

      Instance Name: CalculateWeightedNormalizedBenzeneConcentration
      Instance of Class: Task
      behavior:
      soilSamplingExp.benzeneConcentration = soilSamplingExp.benzeneConcentration / skStandard.skBenzene * benzene.contaminationWeight
      objects: skStandard, soilSamplingExp, benzene

      An observation drawn from examining some existing ontology tools, including Protégé, is that the structure for task behavior cannot be formally represented, which results in inconsistent syntax and grammar. Consequently, machine processing of the task behavior is impossible.
   e. This step describes how to specify task priority, which is a type of strategic knowledge. First, an instance of TaskPriority is created and identified as **tp_CalculateWeightedNormalizedBenzeneConcentration**. The "tp_" prefix is used because Protégé does not allow instances to have the same name. Task Priority consists of a task and a priority number. Below is the specification for this task priority:

      Instance Name: tp_CalculateWeightedNormalizedBenzeneConcentration
      Instance of Class: TaskPriority
      priority: 1
      task: CalculateWeightedNormalizedBenzeneConcentration
   f. After a task priority instance has been created, it can be added to an objective. The specification for the DeterminContaminationLevel objective is as follows:

      Instance Name: DetermineContaminationLevel
      Instance of Class: Objective
      taskPriorityList: tp_CalculateWeightedNormalizedBenzeneConcentration

There are two problems with this process for modeling task priority. First, the software tool requirement that the user has to create a separate entity of TaskPriority to link tasks to objectives distracts the users from the real activity of specifying strategic knowledge. The second problem is related to the visualization of the task priority list in the objective. Figure 7 illustrates the problem.

In Figure 1, the task priorities for **SetContaminationLevel** and **CalculateWeightedNormalizedTolueneConcentration** have been added to the **DetermineContaminationLevel** objective. From looking at the taskPriorityList, one might think the tasks are in the following priority:

1. CalculateWeightedNormalizedBenzeneConcentration
2. SetContaminationLevel
3. CalculateWeightedNormalizedTolueneConcentration.

However, the actual priority of tasks is as follows:

1. CalculateWeightedNormalizedBenzeneConcentration
2. CalculateWeightedNormalizedTolueneConcentration
3. SetContaminationLevel.

In a form-based user interface, which is the kind adopted in Protégé, task priorities are displayed in the order that they were added. In graph-based user interfaces, such as Protégé OWLViz or KAON OI-Modeler, there is no order to the display; the user must look at a separate window or screen for the priority. Again, this problem distracts the user for the real process of knowledge modeling.

To summarize in terms of modeling dynamic knowledge, existing ontology tools demonstrate three main weaknesses. First, their inability to enforce a consistent syntax for task behavior makes computer processing of the task behavior impossible. Second, the lack of support for input and output of tasks can result in specification of an incomplete model. Third, the visualization/input fields are not user

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

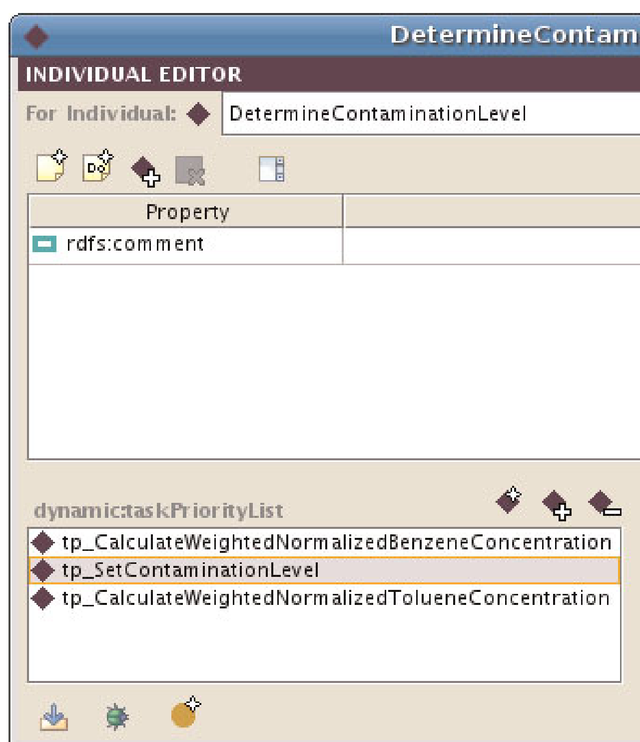

Fig. 1. Determining the contamination level objective with multiple task priorities. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

friendly for enabling the user to easily specify/visualize priority among tasks. Our research work addresses all three weaknesses.

2.3.3. *Ontology testing*

For ontology tools, the third area in need of improvement is support for ontology testing. Software testing is an important part of the software development life cycle because it identifies defects and results in a more stable software application. It is considerably cheaper to fix defects early in the development process. Unit testing is a method of testing the structure of a program. Unit tests can be used as regression tests to ensure that the software continues to work as expected after modifications. Catching and fixing defects in the modeling stage is less expensive and easier than in the implementation stage.

Ontology testing can also help the knowledge engineer develop a more complete model of the domain. In the ontological engineering field, ontology testing is also called ontology evaluation. According to (Gomez-Perez et al., 2005), ontology evaluation should be performed on the following:

- every individual definition and axiom,
- collections of definitions and axioms stated explicitly in the ontology,
- definitions imported from other ontologies,
- definitions that can be inferred from other definitions.

Existing ontology testing systems such as the OWL unit test framework (Horridge, 2005*a*) and Chimaera's (McGuiness et al., 2000) test suite, evaluate the correctness and completeness of ontologies. Chimaera performs tests in the following four areas: missing argument names, documentation, constraints, and so forth; syntactic analysis (occurrence of words, possible acronym expansion); taxonomic analysis (unnecessary super classes and types); and semantic evaluation (slot/type mismatch, class definition cycle, domain/range mismatch; McGuiness et al., 2000). Such testing tools are sufficient for testing static knowledge, but are not suitable for testing the interactions between behavior and objects.

Our research work aims to contribute to the field of ontological evaluation by addressing the difficult issue of testing behavior or dynamic knowledge. Our approach attempts to combine unit testing techniques with the adoption of test cases in test-driven development (TDD; Janzen & Saiedian, 2005). This is a useful hybrid approach for addressing the complex interactions of task behavior and objects. The general intuition adopted from the TDD approach of testing is that it should be "done early and done often." In TDD, a test case is written first and then the actual module is written. Writing test cases first can be beneficial because instead of thinking of test cases as "how do I break something," writing test cases first make you consider "what do I want the program to do." In other words, writing test cases first make you think about what functionality and objects are required. We believe ontology development could also benefit from this kind of approach.

2.3.4. *Ontology management*

The support for ontology management in ontology tools is the fourth area that needs improvement. Ontology development within the SW allows anyone to create, reuse, and/or extend concepts in a distributed, heterogeneous environment. Many different versions of an ontology may be created, resulting in problems of backward compatibility. To document, track, and distribute ontologies in such an environment, an ontology management system is needed. An ontology management system is analogous to a database management system. Weaknesses of existing ontology management systems include vulnerability to malicious ontology authors, no consideration for intellectual property rights, and lack of support for ontology versioning.

2.3.5. *Specific research objectives*

Our survey on existing ontology tools and the weaknesses noted inform our design of a proposed suite of ontology tools. Our design objective is to construct a suite of software tools for modeling and managing knowledge. The tools should have the following characteristics:

1. Provide support for a knowledge modeling technique (we adopted for this purpose, the IMT).
2. Provide support for ontology management.

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

3. Provide support for testing of the developed knowledge model.
4. Provide support for diverse types of knowledge required for developing an ontological model of an industrial domain.

For illustration purposes, we applied our tool for modeling, in this case, the petroleum contamination remediation selection domain. Next the application problem domain is described.

2.4. Application problem domain

2.4.1. Overview of petroleum contamination remediation selection

Petroleum contamination of soil and groundwater is an important environment issue because it can adversely impact the health and well-being of a community and the surrounding natural environment. Petroleum contamination is often the result of leaks and spills from petroleum storage tanks and pipelines, as shown in Figure 2. From the gas tank, contaminants first leak into the top layer of soil, then eventually through the soil, into the lower, groundwater layer. The petroleum contaminants include chemicals such as benzene, toluene, ethyl benzene, xylene, and total petroleum hydrocarbon. These chemicals can potentially cause serious health problems to humans.

The process of cleaning a petroleum-contaminated site is called petroleum contamination remediation. A variety of remediation methods/technologies are available. However, different contaminated sites have different characteristics depending on the pollutants' properties, hydrological conditions, and a variety of physical (e.g., mass transfer between different phases), chemical (e.g., oxidation and reduction), and biological processes (e.g., aerobic biodegradation). Thus, the methods selected for different sites vary significantly. The decision making process for selecting a suitable method at a given site often requires expertise on both the remediation technologies and site hydrological conditions. Because selection process is complex, an automated system for supporting decision making on site remediation techniques is useful. Development of such a decision support system (DSS) can benefit from the use of SW technology like ontology construction.

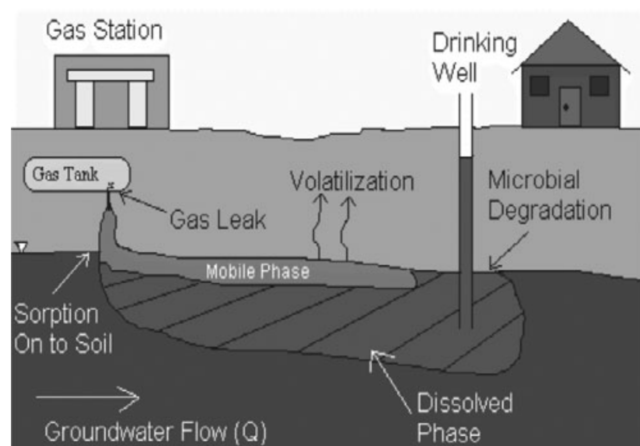

Fig. 2. An overview of the contamination problem caused by petroleum production activities according to Chen (2001).

2.4.2. Details of petroleum contamination remediation selection

There are two categories of remediation techniques: *in situ* and *ex situ. In situ* remediation techniques use treatments on the soil or groundwater; some examples of *in situ* remediation include soil flushing, biostimulation, chemical treatment, and phytoremedizition (Chan et al., 2002). *Ex situ* remediation techniques involve the removal of the contaminated soil by excavation. Some examples of *ex situ* remediation include land treatment, chemical extraction and excavation, air stripping, and carbon adsorption (Chan et al., 2002).

The selection of a remediation technology involves the following five factors (Chen, 2001):

1. Contaminated media: The contaminated media (soil or groundwater) can be either unsaturated or saturated. Not all of the media require cleaning. There are three possible requirements for a remedy: only the unsaturated zone needs to be cleaned, only the saturated groundwater zone needs to be cleaned, or both the unsaturated and saturated zones need to be cleaned.
2. Site hydraulic conditions: The hydraulic properties of a site include the following considerations: soil permeability, site heterogeneity, and isotropism. According to these properties, a site can be classified as simple or complex.
3. Estimated plume size: If the contaminated site size is small, an *ex situ* remediation technique is preferred; otherwise, an *in situ* remediation technique is recommended.
4. Current phase of the immiscible contaminants: If the immiscible contaminants are present in the free phase, then oil recovery has to be considered. If the immiscible contaminants are present in the residual phase, then more complex techniques like integrated remediation technology is needed.
5. Concentration of the contaminants: The concentration of the contaminants can be grouped into the three ranges of low, medium, and high. Different concentration levels of the contamination require different remediation actions.

3. DESIGN AND IMPLEMENTATION OF DYNA

3.1. Overview

To address the objectives of modeling and testing dynamic knowledge, we built a software tool that models dynamic

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

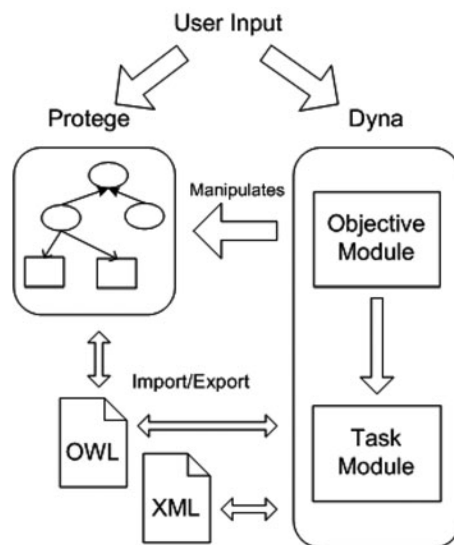

Fig. 3. The architecture of Dyna.

knowledge, called Dyna. Dyna has been built as an extension of Protégé, which is an ontology editing tool created by the Department of Stanford Medical Informatics at Stanford University (Gennari et al., 2003). It is an open-source system programmed in Java, and is very extensible through its plug-in architecture. Dyna is a "Tab Widget" that works with both Protégé-Frames and Protégé-OWL. At the time of writing, Dyna has been tested on Protégé 3.2.1.

A high level view of the architecture of Dyna and its interaction with Protégé is shown in Figure 3. The user creates static knowledge (classes, properties, relations) in Protégé and the dynamic knowledge (objectives and tasks) in Dyna. Dyna contains two main modules for creating objectives and tasks. From the objective module, strategic knowledge can be specified by linking tasks to an objective and prioritizing the tasks. The task module supports the definition of task behavior and the instantiation and manipulation of objects and properties created in Protégé. The task module also supports the creation and running of test cases, a process that helps to verify that the task behavior is working as expected. Both Protégé and Dyna can import and export the models in the OWL file format. Dyna can also import and export the models in the XML file format.

Based on the IMT, the following procedure describes the general process of modeling knowledge with Protégé and Dyna. A detailed description of the process applied to the modeling of a real-world problem domain is presented in Section 4.

1. Model static knowledge in Protégé:
   a. Create a class.
   b. Create a property for the class.
   c. Specify any additional attributes for the class, such as restrictions.
   d. Repeat steps a–c until satisfied that the static knowledge model is complete or the knowledge engineer has enough information for progressing to Step 2 to develop the dynamic knowledge model.
2. Model dynamic knowledge in Dyna:
   a. Create an objective.
   b. Create a task.
      — Specify task behavior.
      — Specify objects (instances of classes) used in the behavior.
      — Specify and run test cases. Object attributes can be modified and checked if they are the expected values.
      — Specify strategy by linking task to objective and setting priority.
   c. Repeat steps a and b until satisfied that the model is complete.
3. Optionally, the static and dynamic knowledge models may be exported in the OWL file format.

3.2. Knowledge representation

The static knowledge components are handled by Protégé, which uses both a Frames-based knowledge model and an OWL-based knowledge model. Both knowledge models provide classes, properties (or slots) of classes, parent–child relations between classes, and individuals (or instances) of classes. Protégé–OWL also supports the many additional knowledge components defined in OWL. See the Stanford Protégé website (http://protege.stanford.edu) for more details on the Protégé knowledge models.

The dynamic knowledge components as defined by the IMT are organized into the object-oriented class hierarchy shown in Figure 4. The top-level component is **KnowledgeComponent**, which defines the properties **name** and **documentation**, which are used for identification and description of a knowledge component. The components **Project**, **Task**, and **Objective** are derived from **KnowledgeComponent**. A **Project** contains a list of tasks and a list of objectives. An **Objective** consists of a prioritized list of tasks required to achieve the objective. A **Task** is an activity that is performed to achieve an **Objective**. A **Task** consists of behavior, input values, one output value, preconditions, objects, and dependencies. A task can also be decomposed into subtasks.

3.3. Objective modeling

Objectives are modeled in an Objective Window, which contains input fields for the name of the objective, documentation, and tasks associated with the objective. The tasks associated with an objective are in a prioritized order so that a task with higher priority should be executed first. Test cases for the tasks can also be run, and tests are run in the order of task priority. This function is discussed further in Section 3.4.

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

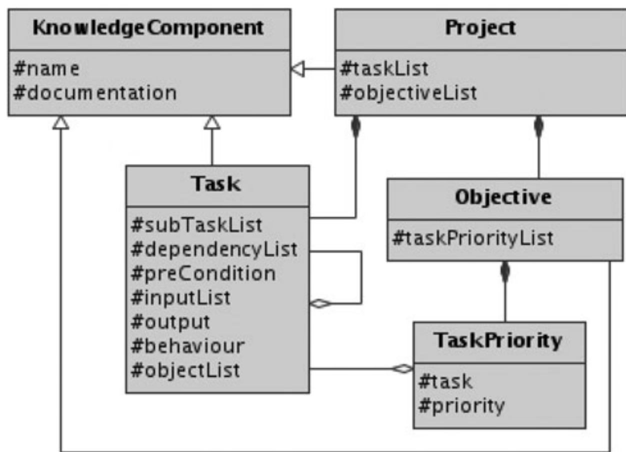

Fig. 4. Dyna knowledge representation.

3.4. Task modeling

Tasks are modeled in a tasks window, which contains input fields for the name of a task, documentation, behavior, objects, dependencies, and test cases of a task. The main components of the task module are the task behavior language (TBL) and its interpreter. Together, TBL and its interpreter enable the specification and manipulation of objects associated with a task and tasks dependent on other tasks. They also enable dynamic knowledge to be tested. These features are described as follows.

3.4.1. TBL and interpreter

A weakness of the KMS (Chan, 2004) is that the system does not support a formal and systematic representation of task behavior. As a result, the representation of task behavior is not formal and involves inconsistent syntax and grammar, which renders machine processing of the task behavior impossible. Dyna solves this problem by using a strictly enforced, formalized and high-level language, called TBL, for representing task behavior.

TBL supports the following basic structures that are common to most programming languages:

- Types: integer, float, Boolean, string
- Mathematical operations: +, −, *, /, %
- Logical operators: and, or, not, xor
- Conditional operators: <, >, ==, <=, >=, !=
- Assignment operator: =
- If statement, While loop, Assert statement, Return statement, Print statement
- Class Object Instantiation
- Function definition and calling
- Comments: #

Dyna enforces the structure of the task behavior with an interpreter, which also enables the task behavior to be run. A high-level view of the architecture of the TBL interpreter is shown in Figure 5. The interpreter consists of a lexical analyzer and a parser, which were generated using JavaCC (https://javacc.dev.java.net/) a tool for generating compilers and interpreters. The lexical analyzer breaks the input task behavior into sequences of tokens. The parser then analyzes the sequences of tokens according to the TBL grammar and generates an abstract syntax tree (AST). If the input task behavior has errors, then the parser outputs an error message. The functionality of each language element was implemented in the AST nodes; and there is an AST node for each TBL language structure. Interpretation of TBL is achieved by performing a depth-first traversal of the AST. As the AST is traversed, encountered symbols or identifiers are stored in the symbol table, values are pushed/popped on/off the stack, and instances of classes in Protégé are modified.

Two questions that might be asked are "why create TBL?" and "why not use an existing language?" Any programming language that supports the basic structures (math operators, loops, conditions, etc.) could have been used for modeling the behavior component of dynamic knowledge. The grammar describing the language, for example C++, would have been processed with a tool similar to JavaCC resulting in a lexer and parser, from which a custom interpreter could be generated. Existing programming languages have many features and language attributes, many of which are not applicable for modeling dynamic knowledge. We believe processing these

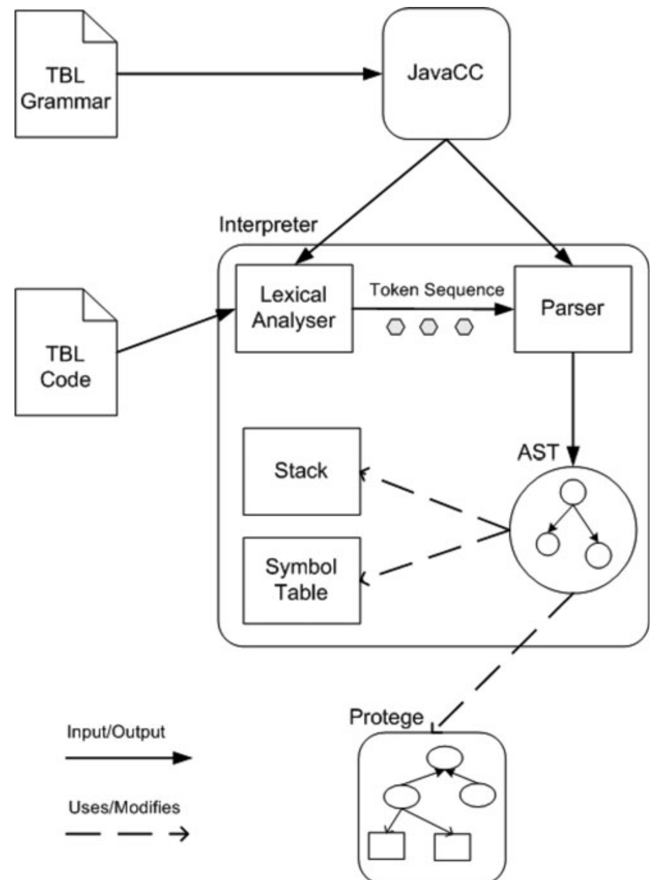

Fig. 5. The design of the TBL interpreter.

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

languages would involve significantly more work as the extra features would have to be navigated and dealt with in some manner. Hence, it would be easier and more efficient to create a simple language and interpreter built specifically for processing task behavior. In contrast, there are nonprogramming languages that could also have been used, for example, the SW Rule Language (SWRL; http://www.w3.org/Submission/SWRL/). SWRL is a combination of OWL and Horn-like rules. SWRL can be used to model processes (Happel & Stojanovic, 2006), so it may be possible to adapt SWRL to model tasks and objectives. SWRL would be a radically different approach to modeling task behavior than the approach adopted by KMS, the system on which Dyna is based. Because the objective of Dyna was to improve upon KMS, and not a different approach, Dyna adopted TBL, which is like a programming language, for modeling task behavior.

3.4.2. Task objects

Task behavior consists of calculations and operations on static knowledge. The Task Window enables the user to select a Class in Protégé to instantiate and specify a name for the resulting object. Operations and calculations can be performed on the object and its properties using TBL. For example, *car.color = "red"* where *car* is an instance of a Car class and its property *color* is set to "red." When a test case is run through the interpreter, the defined operations and calculations on the object are actually performed.

3.4.3. Task dependencies

Tasks can have interactions with other tasks. When a task requires a second task to perform its behavior, this second task is considered to be a "dependency" of the first task. Task dependencies are represented in TBL by: *DependentTaskName()*. Calls to dependent tasks are similar to function calls found in common programming languages. Values can be passed to the dependent task via input parameters. An output value can also be returned from the dependent task. When a dependency is encountered by the interpreter, the dependency is first added to the task's list of dependencies, then the interpreter evaluates the behavior of the dependency.

3.5. Testing dynamic knowledge

TDD is a very useful technique for dynamic knowledge modeling. Writing test cases for tasks first can help the ontology author derive which classes and properties are required in the task behavior, resulting in a more complete model. However, because writing test cases first may initially prove difficult for some users, Dyna supports the creation of test cases at any time during the ontology development process.

Dyna's testing framework is based on JUnit (http://www.junit.org), a unit testing framework for Java. To test a class in JUnit, a test class is created, which contains methods for each test case and a special method called setup() for doing any initialization. Dyna's testing framework is attached to the tasks, because the tasks contain the testable behavior. Each task contains a test suite, which provides a "setup" module and facilities for creating test cases and defining them in TBL. Test cases require that the "assert" function be called; and the assert function takes as a parameter a condition that is necessary for the test case to succeed. When the test case is run through a test interpreter, the interpreter first performs any necessary initialization, then it executes the behavior of the task, and evaluates the condition of the assert function. Depending on whether the assert function returns a "true" or "false" value, the interpreter would display a message notifying the user if the test case has "passed" or "failed."

3.6. Knowledge storage and interoperability

3.6.1. XML

Dyna projects are stored in the XML file format, enabling them to be shared and reused by other systems. Most of the dynamic knowledge components and their properties are represented as a hierarchy of XML elements. The following is an excerpt from the XML representation of a task:

```
<Task>
     <Name/>
     <Documentation/>
     <SubTaskList/>
     <DependencyList/>
     <Behavior/>
     <ObjectList>
          <Object/>
     </ObjectList>
     <PreCondition/>
     <TestSuite>
          <TestSetup/>
          <TestCaseList/>
     </TestSuite>
</Task>
```

The class objects that have been instantiated in Dyna require a link to their definitions in Protégé. Protégé can save its data, including the class object definitions, to an OWL file. OWL is an XML-based SW technology. Another XML-based SW technology, RDF, has the ability to create links to data defined in other XML-based files. An example of using RDF to link an object in a Dyna XML file to its definition in an OWL file is as follows:

```
<Object
rdf:resource="http://example.com/petrol_rem_sel.owl#Benzene_B"/>
```

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

where *rdf:resource* defines where the definition of this object can be found. The static knowledge is stored in the OWL file, *petrol_rem_sel.owl* on a hypothetical web server at http://example.com. *Benzene_B*, is the identifier of the object.

```
<owl:Class rdf:about="#Task">
     <rdfs:subClassOf rdf:resource="#KnowledgeComponent"/>
</owl:Class>
<owl:FunctionalProperty rdf:ID="name">
     <rdf:type
rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
     <rdfs:domain rdf:resource="#KnowledgeComponent"/>
     <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
   </owl:FunctionalProperty>
```

3.6.2. OWL

A weakness of Dyna XML is that it lacks semantics, making it difficult to use the dynamic knowledge in other systems. To achieve true interoperability with other systems, Dyna projects can be exported to OWL, which provides semantics to XML. In this way, it is possible for another system to know, for example, that a Task is an instance of a task in the dynamic knowledge model.

The modeling of the dynamic knowledge elements was done in Protégé-OWL. There are three types of OWL to choose from: OWL Lite, OWL DL, and OWL Full. OWL Lite is the simplest, but least expressive. OWL DL is more expressive than OWL Lite. OWL Full is the most expressive, but performing reasoning on it is very difficult. The type of OWL chosen for the project was OWL DL because it is sufficiently expressive, and the purpose of modeling the petroleum contamination remediation selection domain is to eventually create a DSS, which means automated reasoning is required. OWL DL proved to be indeed expressive enough, and every dynamic knowledge element in the remediation domain was represented. The final OWL dynamic knowledge model was verified as valid OWL DL by using the WonderWeb OWL Ontology Validator (http://phoebus.cs.man.ac.uk:9999/OWL/Validator). Because the model is in valid OWL DL, any other system that can read OWL DL, will be able to use this model.

The following is a sample of the OWL code used to model the dynamic knowledge components. The OWL in this example defines a class **Task** and states that it is a subclass of **KnowledgeComponent.** There is also a definition of the property **name**.

Dyna's projects are exported to OWL simply by traversing the internal knowledge model and outputting each element's OWL representation to a file. The following is a sample of the OWL code for representing a task; the sample task is for deciding the number of sampling points. **dyn:Task** defines an instance of the **Task** class, with the **name** property set to "DecideNumberOfSamplingPoints."

```
<dyn:Task rdf:ID="DecideNumberOfSamplingPoints">
<dyn:name
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">DecideNumberOfSamplingPoints
</dyn:name>
</dyn:Task>
```

Dyna's exported OWL projects have also been verified as valid OWL DL by the WonderWeb OWL Ontology Validator.

4. APPLICATION OF DYNA FOR DEVELOPING AN ONTOLOGY MODEL OF PETROLEUM CONTAMINATION REMEDIATION SELECTION DOMAIN

4.1. Overview

Protégé and Dyna were applied to create an ontology of the petroleum contamination remediation selection domain. The process of selecting a remediation technology for a petroleum contaminated site involves many steps that interact with a number of different objects. According to the IMT, knowledge is considered to be either static or dynamic; therefore, the knowledge elements in the petroleum contamination remediation selection domain are categorized into the two types/groups of knowledge. Static knowledge includes concrete objects such as soil, water, and contaminants, which are modeled using Protégé-OWL 3.2.1 (described in Section 4.2). Dynamic knowledge includes objectives, such as "Select Remediation Method" and actions or tasks that are required to achieve the objective. The dynamic knowledge is modeled in Dyna and is described in greater detail in Section 4.3.

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

4.2. Static knowledge

The static knowledge of the Petroleum Contamination Remediation Selection domain was implemented in Protégé-OWL 3.2.1 using an existing ontology developed by (Chen, 2001) with the assistance of an environmental engineer. Again, OWL DL was adopted because of its expressiveness and its support of automated reasoning. The static knowledge was modeled using an iterative process of creating a class and then its properties and other characteristics. The process of class creation will be described with an example. Because the **media** (soil, water, groundwater) that has been contaminated is an important knowledge element in the Petroleum Contamination Remediation Selection domain, this element is used in the following example for illustrating the process of creating a class and other characteristics:

1. Create a class: **Media**
2. Create a property for the class. OWL supports two types of properties: datatype and object. Datatype properties are for properties of simple types such as "integer" or "string." Object properties are for properties that are individuals of other classes. The size of Media can be classified as "small," "medium," or "large." Because these are strings, a datatype property identified as **siteSize** was created.
   a. Specify the domain(s) of the property. For **siteSize**, the domain was set to **Media**.
   b. Specify the range of the property. The possible values for **siteSize** ("small," "medium," "large") are all strings, so the range of **siteSize** was set to **string**.
   c. Specify restrictions for the property.
      — Restrict the possible allowed values. The only possible values for the size of a site are "small," "medium," and "large." To enforce this restriction, these values were input into the Allowed Values field.
      — Specify whether or not the property is Functional. OWL individuals (instances) can have multiple values of a property, which means that there can be multiple values of a property for an individual. If a property is limited to one value for an individual, then that property is "functional." An individual of **Media** can only have one value of **siteSize** at a time (a medium cannot be both "small" and "large"), so **siteSize** is set to "functional."
3. Repeat steps 1 and 2 until either all the classes are finished or the knowledge engineer has enough information to develop the dynamic knowledge model.
4. Specify disjointness. OWL individuals can be of more than one type. For example, it is possible to specify that an individual is both a **Media** and a **Remediation**. Such cases of multiple inheritance can cause unforeseen negative consequences. To prevent such modeling errors, a Class can be made disjoint with one or more other classes. In the case of the **Media** class, it is disjoint with all of its sibling classes (**Contaminant**, **Experiment**, **Gas**, **Mathematics**, and **Remediation**).

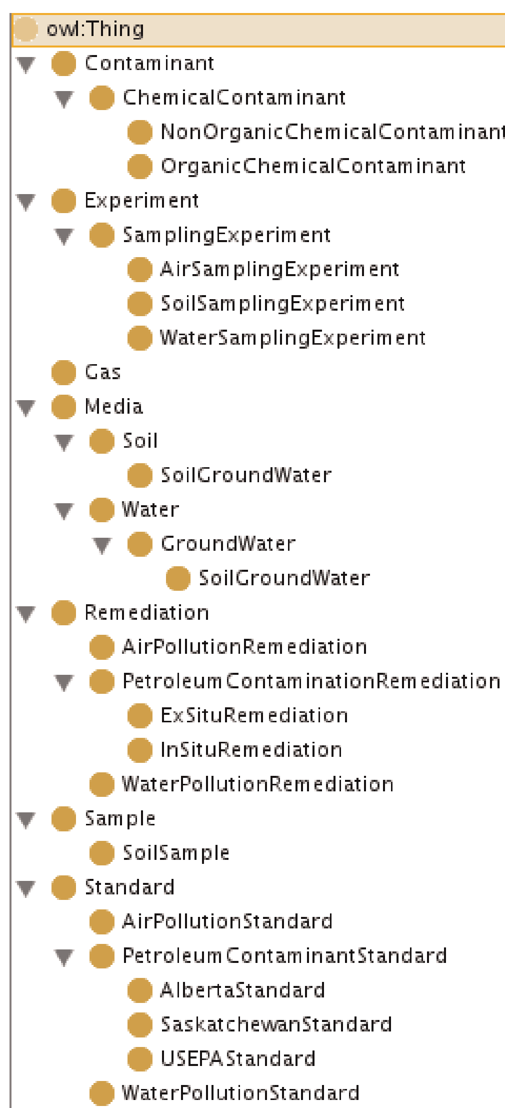

Fig. 6. The static knowledge class hierarchy in Protégé. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

Shown in Figure 6 is the class hierarchy of the static knowledge.

4.3. Dynamic knowledge

The dynamic knowledge was constructed in Dyna using an existing ontology discussed in Chen (2001). An iterative process of creating an objective, followed by its tasks was used. An example of dynamic knowledge for the Petroleum Contamination Remediation Selection domain is determining the classification of the size of a site (or media). Site size

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

determination is used in the following to show the process of creating objectives and tasks:

1. Create an objective: **DetermineSiteSize**
2. Create a task: **SetSiteSize**

    a. Specify task behavior and any inputs and output. The behavior for **SetSiteSize** is as follows:

```
Inputs: int area, int volume
if (area < 1600 and volume < 25000)
{
   site.siteSize = "small"
}
else if (area >= 1600 and area <= 2000 and
         volume >= 25000 and volume <= 30000)
{
   site.siteSize = "medium"
}
else if (area > 2000 and volume > 30000)
{
   site.siteSize = "large"
}
```

This task behavior is input into the Task Behavior screen shown in Figure 7.

    b. Specify any objects, that is, instances or individuals of classes, which are used in the behavior. **SetSiteSize** uses the object **site**, which is an individual of **Media**.
    c. Specify test cases. The test case for verifying that the size of a site is "small" is as follows:

```
site.siteArea = 1000
site.siteVolume = 1000
SetSiteSize(site.siteArea, site.siteVolume)
assert(site.siteSize == "small")
```

The test case begins with setting the area and volume of the site. Then the task, **SetSiteSize**, is called and its behavior is run through the TBL interpreter. The TBL interpreter will evaluate the condition of the first "if" statement and determine that the condition is true and then the interpreter will set the value of site.siteSize to "small." When the TBL interpreter is finished interpreting **SetSiteSize**, the size of the site is verified with the assert function. In this case, the assertion will be "true;" therefore, the following message will be output to a screen:

```
SetSiteSize
     testSmall : Passed
```

If the assertion was "false," the message output to a screen would be:

```
SetSiteSize
     testSmall : Failed
```

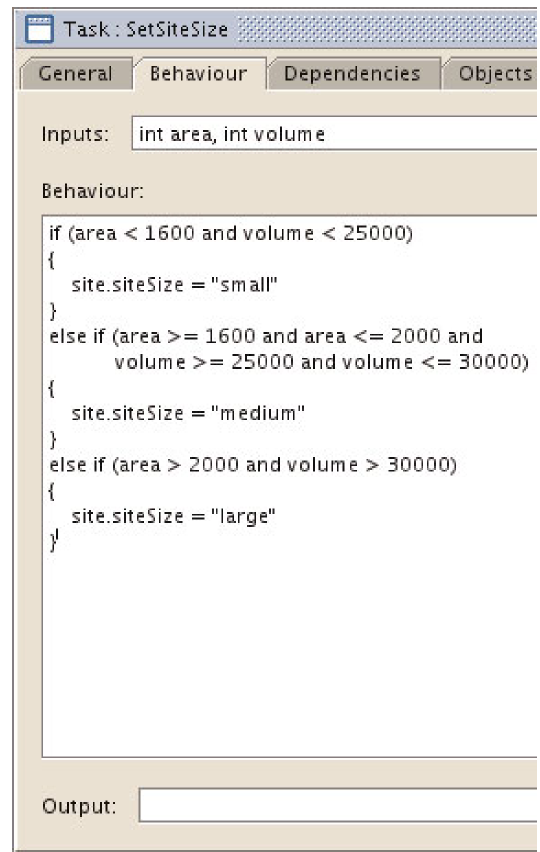

Fig. 7. Dyna: task behavior. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

The testing suite, shown in Figure 8, provides the facilities for adding, deleting, and running test cases. Test cases may be run individually or together. The test cases shown in Figure 8 are for the task of **SetSiteSize**. The test case that is highlighted is for testing of a "small" site size, which was described above.

3. Link the task to the objective. The link between the task **SetSiteSize** and the objective **DetermineSiteSize** is shown in Figure 9.
4. Adjust the priority of the task in the objective, if necessary. **SetSiteSize** has a priority of 3, which means that it is to be performed after the tasks **MeasureSiteArea** and **MeasureSiteVolume**.
5. Repeat Steps 1–4 until the knowledge engineer is satisfied that the model is complete. Although it is difficult to know when the model is complete, a useful heuristic is to assess whether all the tasks have test cases and if they run successfully. If both criteria are satisfied, then it is likely the model is complete.

Most of the petroleum contamination remediation selection ontology was successfully modeled. The root objective is **SelectRemediationMethod**; all the other objectives and tasks fall under this objective. **SelectRemediationMethod** is described in Table 1 to briefly discuss the final ontology.

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

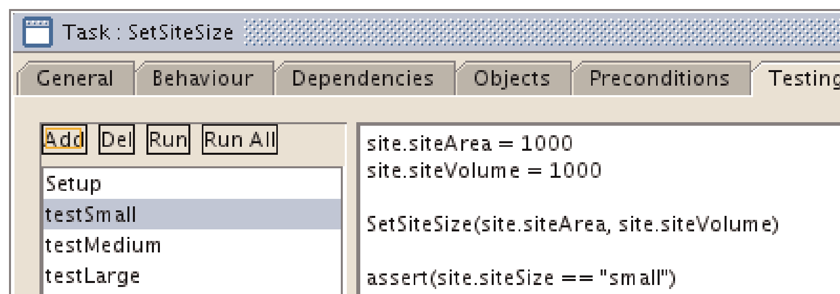

Fig. 8. Dyna: testing task behavior. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

The first task, **DetermineSiteSize**, measures the area and volume of the site and then classifies the size of the site. The **SoilSampling** task involves a number of different activities such as collecting soil samples, measuring thicknesses and percentages of the soil types in the soil samples, and measuring contaminant concentrations. **DetermineSiteHydraulicConductivity** finds the degree to which water can move between the spaces between the particles of soil. **DetermineContaminationLevel** finds the contamination level by calculating the concentrations of benzene, ethyl benzene, toluene, and xylenes. **DetermineContaminationPhase** simply sets the phase as "free" or "residual." The final task, **SetRemediationMethod**, considers the following five factors, which are found in the preceding tasks, to determine the appropriate remedial technology:

1. Media type (soil or soilgroundwater)
2. Hydraulic conductivity (simple or complex)
3. Site size (small or large)
4. Contaminant phase (free or residual)
5. Contamination level (low or high).

All of the static knowledge was successfully implemented in Protégé and most of the dynamic knowledge was successfully implemented in Dyna. There are some limitations of Dyna, which prevented some knowledge from being modeled. These limitations are discussed further in Section 5. There were a few tasks, whose behavior could not be modeled due to lack of information. The values for soil texture contribution factor were stored in a database; there were no calculations on how the values were derived. Therefore, for testing purposes a value for soil texture contribution factor has been hard coded into the behavior of the task for calculating soil texture contribution factor.

5. DISCUSSION

During the implementation of the petroleum contamination remediation selection ontology in the Class Editor and Dyna, a number of interesting observations were made. While discussing these observations it is important to keep in mind that at the time of writing the tools have only been tested on

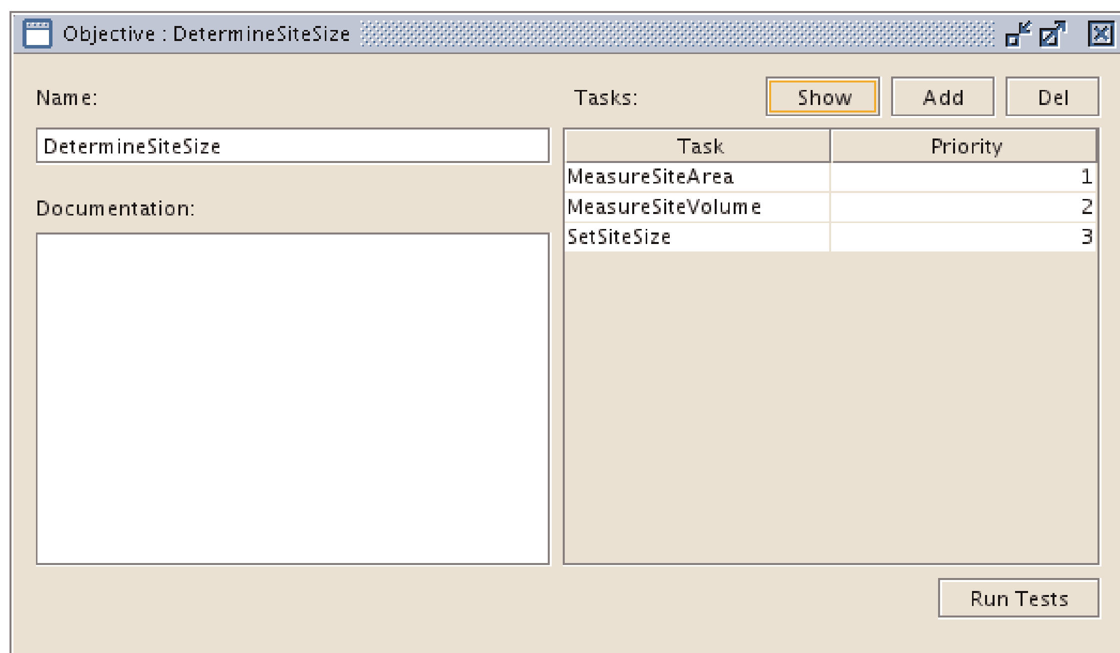

Fig. 9. Dyna: DetermineSiteSize objective. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

Table 1. *Select remediation method objective*

Objective	Tasks
SelectRemediationMethod	1. DetermineSiteSize 2. SoilSampling 3. DetermineSiteHydraulicConductivity 4. DetermineContaminantionLevel 5. DetermineContaminationPhase 6. SetRemediationMethod

a single domain of knowledge. Most of these points may be applicable when the tool is applied to other domains.

A number of observations can be drawn about Dyna from the process of modeling the petroleum contamination remediation selection ontology; some comments can be made about the modeling process itself. The following presents some observations about the modeling process, and some advantages and weaknesses of Dyna.

In modeling with the IMT, a distinction is made between a "task" and an "objective." High-level operations are objectives (e.g., **SelectRemediationMethod**) and operations with specific actions are tasks (e.g., **MeasureSandThickness**). Tasks can be identified using keywords that describe simple actions, such as "set," "measure," or "add" and keywords that indicate more general or higher level action words, "determine" or "find" can be used for identifying objectives. However, when this distinction is applied for modeling the remediation domain, it became apparent that the general guideline that objectives are high-level operations and tasks are specific actions cannot always be followed. For example, the original knowledge table in (Chen, 2001) listed the objective "Select Remediation Method" as containing the task "Determine Site Size." Following the guideline, "Determine Site Size" is an objective as it is a high-level operation that contains a few different actions. The ideal solution would be to add the "Determine Site Size" objective to the list of tasks of "Select Remediation Method," but this is not possible as the knowledge representation in Dyna does not support this. The solution used was to create a task called **DetermineSiteSize**. The behavior for such a task duplicates the objective's prioritized tasks. This solution is not satisfactory because the task behavior duplicates the objective's prioritized tasks.

Using the correct names for tasks and objectives is very important as it greatly affects the intended meaning of the model. Many of the tasks are "Measure" tasks, such as **MeasureBenzeneConcentration**. These "Measure" tasks set the value of some property for an object. In the case of **MeasureBenzeneConcentration**, the **SoilSamplingExperiment** object's **benzeneConcentration** value is set. This is typical behavior of a "setter" function, which is used to set the value of a member variable for a class. Therefore, instead of **MeasureBenzeneConcentration**, we could have **SetBenzeneConcentration**. However, "Set" is not a suitable prefix in this situation. "Set" and "Measure" have different meanings. "Measure" implies that we are performing some action to find a value. In contrast, "set" implies that we already know the value, and are simply recording it.

The IMT has suggested that a knowledge engineer should model static knowledge first, and when the static knowledge model has been developed, then the dynamic knowledge is modeled. However, we have found this suggested sequence of modeling is not always practical. Often, the ontology author is uncertain what classes and properties are required. As a consequence, creating static knowledge first leads to errors, such as incorrectly named objects and properties attached to the wrong classes. From our experience, we found a more fruitful approach is to use an iterative process of creating some dynamic knowledge first, and using it as a guide to create the static knowledge required to ensure the dynamic knowledge can be implemented. This is similar to the process of writing test cases first in TDD, described in Section 2.3.3. By creating tasks and objectives first, the user has to think about what functionality and objects are required, which greatly reduces the amount of errors when objects are specified.

The two main components that were most helpful in revealing missing information and modeling errors were the fields for task behavior input and output and dynamic knowledge testing. Both components were helpful in revealing new classes and properties that need to be added to the static knowledge. The fields for task behavior input and output can be used for prompting the user to think about the relationships between tasks and the objects manipulated by the tasks. The testing of dynamic knowledge can be used in identifying new classes and properties that need to be added to the static knowledge. The creation of test cases can also reveal logical errors in the task behavior. For example, an erroneous "if statement" for classifying site size was as follows:

```
if (area < 1600 and volume < 25000)
{
    site.siteSize = "small"
}
else if (area >= 1600 and area < 2000
and volume >= 25000 and volume <= 30000)
{
    site.siteSize = "medium"
}
else if (area > 2000 and volume > 30000)
{
    site.siteSize = "large"
}
```

The logical error is in bold and underlined in the above behavior. If **area** equals 2000, then the size of the site will not be set. Running the test cases for this task would generate a failure, thereby alerting us of the error in the model. When the error is corrected, the test cases should all run successfully. Dynamic ontology testing facilitated verification and enabled creation of a more complete knowledge model.

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

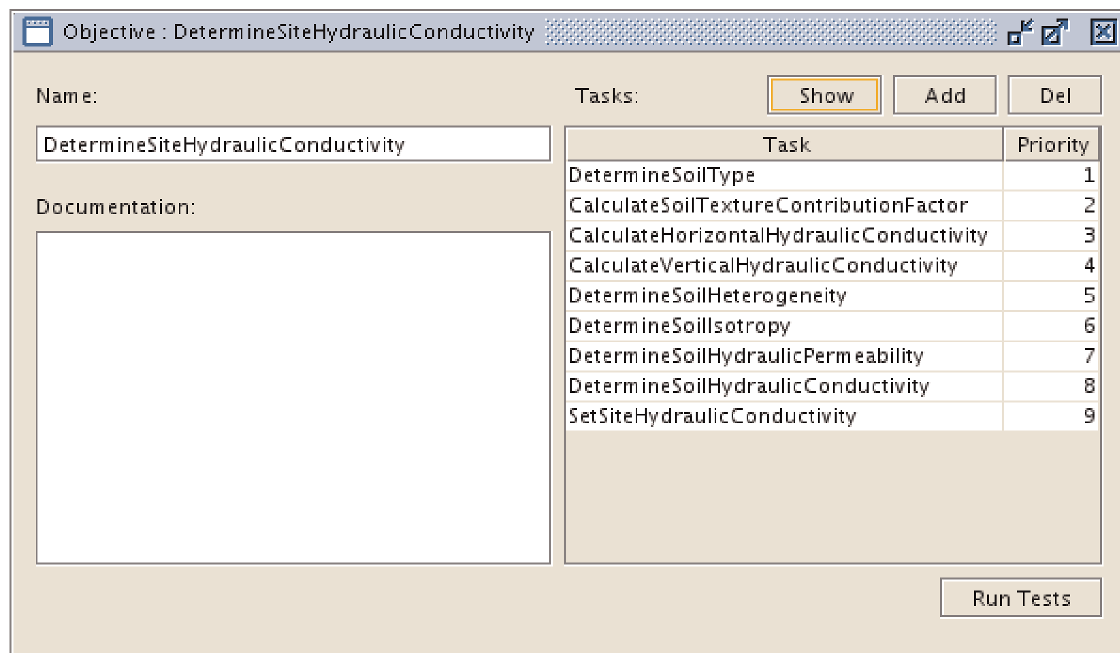

Fig. 10. Objective: DetermineSiteHydraulicConductivity. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

Not all of the dynamic knowledge in the original ontology could be modeled in Dyna. The main weakness of Dyna is its lack of support for the array data type. Many of the tasks perform operations on groups (or arrays) of objects. For example, one of the tasks measures the percentage of sand for each soil sample in the soil sampling experiment. This task requires a loop that iterates through one or more soil samples. However, because Dyna does not support arrays, only one soil sample can be measured. Therefore, in implementing the ontology model, it is assumed that the number of sampling points equals one (numSamplingPoints = 1), and there is one soil sample (soilSample1). The lack of support for arrays also made it impossible to select multiple remediation methods. Under certain conditions more than one remediation method can be selected. However, without arrays, it is not possible to model those conditions.

Another weakness of Dyna is its lack of support for comprehensive visualization of objectives and tasks. Objectives and tasks occur in a sequence or flow; however, the forms-based visualization that is currently used makes the sequence or flow difficult to understand. Figure 10, shows the objective **DetermineSiteHydraulicConductivity** and its prioritized list of tasks. The prioritized list of tasks is a sequence. A visualization of the sequence of tasks and objectives can give the user a better understanding of the interactions between the different objectives and tasks. It is difficult for the user to picture the entire sequence of tasks and objectives because of two problems with the visualization. First, the view of the sequence is limited to a single objective. In Figure 10, for example, **DetermineSoilType** is followed by **CalculateSoilTextureContributionFactor**. This sequence is missing tasks of the objective **DetermineSoilType**. The second problem is that the there is no context for which the objective takes place in. In Figure 10, the screen does not show that **DetermineSiteHydraulicConductivity** takes place within the context of the objective **SelectRemediationMethod** or that the objective **SoilSampling** and its tasks precede **DetermineSiteHydraulicConductivity**. Because of these two problems the user is required to picture the sequence of tasks and objectives in their mind, thus making the modeling process more difficult. A graphical view that can show the full sequence of objectives and tasks would be very beneficial.

The use of TBL requires programming experience. TBL is not intended for use by domain experts, because they may not have expertise in programming. Only the knowledge engineers, who should have programming experience, are intended to use TBL. The domain expert is the source of knowledge. The knowledge engineer acquires knowledge from the domain expert and represents the acquired knowledge in a model specified in TBL. A change in the domain knowledge is likely to be initiated by the domain expert, who must inform the knowledge engineer. Then the knowledge engineer is responsible for implementing the changes in the model.

The lack of support for ontology management in Dyna is also a weakness. The task behavior is dependent on the classes and properties defined in the static knowledge model. The static knowledge cannot be modified or removed without having an effect on the dynamic knowledge. If a static knowledge element, such as a class, is deleted from Protégé, there is no notification to the user in Dyna that the knowledge element has been deleted. The user may not realize that the

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

knowledge element has been removed until the test cases are run and error messages alert the user regarding an "undefined" knowledge element. An ontology management system can alert the user to the effect their modifications on the static knowledge will have on the dynamic knowledge and possibly prevent any possible problems that will result from their changes.

6. CONCLUSIONS

Sharing and reusing knowledge can help reduce the high costs of constructing KBSs. Technologies and ideas from the emerging SW can be useful for developing KBSs. One of these technologies is the ontology, which can be defined as a shareable, computer processable model of a domain of knowledge. However, there are difficulties in constructing ontologies.

The objective of the research work presented in this paper is to construct a software tool for modeling knowledge for the SW. For modeling dynamic knowledge, Dyna has been developed by extending Protégé so as to render it capable of supporting IMT. A new language called TBL has been proposed for representing task behavior. TBL supports formal expression of task behavior so that task knowledge defined in an ontology can be more shareable and reusable. The dynamic knowledge model can be tested by running test cases on the task behavior. The dynamic knowledge is stored in an XML format, but can also be exported to OWL so that it can be shared and reused by other systems.

Protégé and Dyna have been applied for creating an ontology in the petroleum contamination remediation selection domain. Because Dyna supports testing of the dynamic knowledge model, the test cases generated enable creation of a more complete knowledge model.

7. FUTURE WORK

7.1. Knowledge evolution

The software tool described in this paper is part of a set of tools (Harrison & Chan, 2007). Development of an ontology management system, called the Distributed Framework for Knowledge Evolution (DFKE), is currently ongoing (Obst, 2006). DFKE uses a peer-to-peer (P2P) architecture to share ontologies. Within the P2P network, ontologies are shared using an internal knowledge representation format. The DFKE supports the export of ontologies in its P2P network to XML. DFKE addresses problems of vulnerability of ontologies to malicious authors and the lack of consideration for intellectual property rights. Support for ontology versioning is under development within the DFKE. An interesting feature of the versioning component is that it enables the user to add new concepts from other ontologies into the one under development.

Because Dyna does not currently support ontology management, a possible direction for future work is to integrate DFKE with Dyna. However, the integration is challenging in two aspects. The first challenge is the diverse knowledge representation schemes adopted because DFKE and Dyna, which is built on Protégé, use different knowledge representation mechanisms for both static and dynamic knowledge. The second challenge involves compatibility and communication between Dyna and DFKE because Dyna has been built on Protégé, which is implemented in Java, while DFKE is implemented in Python. These challenges remain on the future research agenda.

7.2. Evaluation of Dyna

A preliminary evaluation should be performed by comparing the completeness of the petroleum contamination remediation selection ontology created in Dyna to the same ontology created in KMS. Further research is required to find an appropriate formal evaluation method for assessing the ontology tool.

ACKNOWLEDGMENTS

We are grateful for the generous support of research grants from the Canada Research Chair Program and Natural Sciences and Engineering Research Council (NSERC). We also thank Zhiying Hu for her help in clarifying knowledge in the ontology model of the domain of selection of a remediation technology for a petroleum contaminated site.

REFERENCES

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American 2001 (May)*, 35–43.

Chan, C.W. (2004). From knowledge modeling to ontology construction. *International Journal of Software Engineering and Knowledge Engineering 14(6)*.

Chan, C.W., Huang, G., & Hu, Z. (2002). *Development of an Expert Decision Support System for Selection of Remediation Technologies for Petroleum-Contaminated Sites*. Rep. No. 00-03-018. Regina, Canada: University of Regina, Petroleum Technology Research Center.

Chen, L. (2001). *Construction of an ontology for the domain of selecting remediation techniques for petroleum contaminated sites*. MSc Thesis. University of Regina, Canada.

Cranefield, S., Pan, J., & Purvis, M. (2000). A UML ontology and derived content language for a travel booking scenario. In *Ontologies for Agents: Theory and Experiences* (Tamma, V., et al., Eds.), pp. 259–276. Basel: Birkhäuser Verlag.

Fensel, D., Harmelen, F. van, Ding, Y., Klein, M., Akkermans, H., Broekstra, J., Kampman, A., Meer, J. van der Studer, R., Sure, Y., Davies, J., Duke, A., Engels, R., Iosif, V., Kiryakov, A., Lau, T., Reimer, U., & Horrocks, I. (2002). On-to-knowledge in a nutshell. *IEEE Computer*.

Gabel, T., Sure, Y., & Voelker, J. (2004, April 7). *KAON—An Overview*. Technical Report, University of Karlsruhe, Insitute AIFB.

Gal, A., Eyal, A., Roitman, H., Jamil, H., Anaby-Tavor, A., Modica, G., & Enan, M. (2006). OntoBuilder. Accessed at http://iew3.technion.ac.il/OntoBuilder/

Gasevic, D., Djuric, D., & Devedzic, V. (2005). Ontology modeling and MDA. *Journal of Object Technology 4(1)*, 109–128.

Gennari, J.H., Musen, M.A., Fergerson, R.W., Grosso, W.E., Crubezy, M., Eriksson, H., Noy, N.F., & Tu, S.W. (2003). The evolution of Protégé: an environment for knowledge-based systems development. *International Journal of Human–Computer Studies 58(1)*, 89–123.

Gomez-Perez, A., Fernandez-Lopez, M., & Corcho, O. (2003). WebODE ontology engineering platform. Accessed at http://webode.dia.fi.upm.es/WebODEWeb/index.html

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press

Gomez-Perez, A., Fernandez-Lopez, M., & Corcho, O. (2005). *Ontological Engineering: With Examples From the Areas of Knowledge Management, e-Commerce, and the Semantic Web*. Berlin: Springer.
Gruber, T. (1993). Towards principles for the design of ontologies used for knowledge sharing. In *Formal Ontology in Conceptual Analysis and Knowledge Representation* (Guarino, N., & Poli, R., Eds.). Padua, Italy: Kluwer.
Guarino, N. (1998). Formal ontology in information systems. *Proc. 1st Int. Conf. Formal Ontology in Information Systems (FOIS'98)*, pp. 3–15. Amsterdam: IOS Press.
Harrison, R., & Chan, C.W. (2005). Implementation of an application ontology: a comparison of two tools. *Artificial Intelligence Applications and Innovations II:2nd IFIP TC12 and WG12 Conf. Artificial Intelligence Applications and Innovations (AIAI-2005)*, pp. 131–143, Beijing, September 7–9.
Harrison, R., & Chan, C.W. (2007). Tools for industrial knowledge modeling. *Proc. 20th Annual Canadian Conf. Electrical and Computer Engineering (CCECE'07)*, Vancouver, April 22–26.
Horridge, M. (2005*a*). OWL unit test framework. Accessed at http://www.co-ode.org/downloads/owlunittest/
Horridge, M. (2005*b*). Protégé OWLViz. Accessed at http://www.co-ode.org/downloads/owlviz/co-ode-index.php
Janzen, D., & Saiedian, H. (2005). Test driven development: concepts, taxonomy, and future direction. *IEEE Computer 38(9)*, 43–50.
McGuiness, D., Fikes, R., Rice, J., & Wilder, S. (2000). An environment for merging and testing large ontologies. *Proc. KR 2000*, pp. 485–493.
McGuiness, D., Fikes, R., & Feigenbaum, E. (2003). Ontolingua. Accessed at http://www.ksl.stanford.edu/software/ontolingua/
Obst, D. (2006). Distributed framework for knowledge evolution. *University of Regina Graduate Student Conf.*, Regina, SK, Canada.

Robert Harrison is currently a Consultant and Software Developer at Online Business Systems in Calgary. Robert is a graduate student at the University of Regina and recently completed the requirements to receive an MASc degree in electronic systems engineering. Robert has a BSc degree in computer science from the University of Regina.

Christine W. Chan is currently Canada Research Chair Tier 1 in Energy and Environmental Informatics and Professor of engineering in software systems engineering in the Faculty of Engineering of the University of Regina. She received MSc degrees in computer science and management information systems from the University of British Columbia and a PhD degree in applied sciences from Simon Fraser University. Dr. Chan is an Editor of *Engineering Applications of Artificial Intelligence* and Associate Editor of *International Journal of Cognitive Informatics and Natural Intelligence* and *Journal of Environmental Informatics*. She is an associate member of the Laboratory of Theoretical and Experimental Philosophy at Simon Fraser University and an adjunct scientist of Telecommunications Research Laboratories.

https://doi.org/10.1017/S0890060409000109 Published online by Cambridge University Press