From: AAAI-02 Proceedings. Copyright © 2002, AAAI (www.aaai.org). All rights reserved.

A Web-based Ontology Browsing and Editing System

Jérôme Thoméré[1], Ken Barker[3], Vinay Chaudhri[1], Peter Clark[2], Michael Eriksen[1], Sunil Mishra[1], Bruce Porter[3] and Andres Rodriguez[1]

[1]SRI International
333 Ravenswood Ave
Menlo Park, CA 94025
{thomere, acr, chaudhri, smishra, eriksen}
@ai.sri.com

[2]Boeing Research and Technology
P.O. Box 37070
Seattle, WA 98124
clarkp@redwood.rt.cs.boeing.com

[3]University of Texas
At Austin
Austin, TX 78712
{kbarker, porter}@cs.utexas.edu

Abstract

Making logic-based AI representations accessible to ordinary users has been an ongoing challenge for the successful deployment of knowledge bases. Past work to meet this objective has resulted in a variety of ontology editing tools and task-specific knowledge-acquisition methods. In this paper, we describe a Web-based ontology browsing and editing system with the following features: (a) well-organized English-like presentation of concept descriptions and (b) use of graphs to enter concept relationships, add/delete lists, and analogical correspondences. No existing tool supports these features. The system is Web-based and its user interface uses a mixture of HTML and Java. It has undergone significant testing and evaluation in the context of a real application.

Keywords: knowledge acquisition, expert systems, HTML, WWW.

Introduction

Our goal is to develop tools that enable domain experts to build knowledge bases (KBs) without relying on AI scientists and engineers. The KBs we envision are ones that support automated reasoning, not just string matching and information retrieval. That is, the KBs are written in a logical formalism, not a natural language.

Achieving this goal is essential for the efficacy of knowledge-based systems. Decades of research on knowledge representation and reasoning have produced the basic technology for knowledge-based systems, so knowledge capture is evermore the bottleneck. Three invariants are responsible for this bottleneck: significant KBs are required, domain experts lack the knowledge engineering skills to build them, and knowledge engineers lack the domain expertise to replace the experts.

Tools can help, and many have been built. Frame-based representation systems have included specialized editors such as Protégé (Noy, Sintek et al. 2001), OntoEdit (Maedche 2000), Ontosaurus (Swartout, Patil et al. 1998), WebOnto (Domingue, Motta et al. 1999), and the Java Ontology Editor (Mahalingam and Huhns 1997). Graphical KB editors, such as the GKB-Editor (Paley and Karp 1996), use directed graphs as the primary means of interaction. Another approach is taken by task-specific knowledge-acquisition systems, such as EXPECT (Blythe, Kim et al. 2001), which is aimed at acquiring process knowledge. However, most of these tools are targeted toward users with significant experience of how knowledge is represented in a knowledge base.

Our research focuses on building a new class of tools for building and using KBs, ones that are used by domain experts with very little training. The new tools should enable domain experts to browse, create/edit, and query KBs—all without knowing logical formalisms and without assistance.

This paper describes one such tool: SHAKEN, built under the auspices of DARPA's Rapid Knowledge Formation (RKF) project. SHAKEN is a collection of modules for different KB interaction tasks: browsing, editing, and querying. Making SHAKEN usable by domain experts presented challenges for each of these modules.

The layout of the paper reflects these three major KB interaction tasks. We also describe an evaluation of SHAKEN by domain experts tasked with building KBs in a domain of biology.

Design Goals

SHAKEN was designed to satisfy the following criteria:

- It should be accessible to users not familiar with any formal logic notation.
- All the operations and interactions should be as simple as possible, and should not require more than a simple introduction.
- The presentation of the concepts, both graphical and textual, should be natural.

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

- The logical representations coming from the knowledge representation system should be transparent to the user.
- The editor should be accessible through a Web browser and should not require any installation at the user end.

Architecture

SHAKEN is implemented as a client-server system. The KB is stored in the Knowledge Machine (KM) representation system (Clark and Porter 1999). KM is object oriented and has the expressive power of full first-order logic. In addition, it supports a STRIPS representation of actions and a situation mechanism to represent different states of a KB. Knowledge analysis and analogy methods are reasoning services implemented on top of KM.

Most of the inference is done on the server side. It includes explanation design plans to describe concepts, analogy methods, and knowledge simulation to test the knowledge entered.

The server functionality is implemented using the allegro server from Allegro Common Lisp. The server is responsible for querying and modifying the KB, managing the interaction between the different components, and presenting the results to the user. The output is presented using HTML and XML. XML is used for representing the concepts, which are then parsed by the Java applet and rendered into graphs, akin to semantic networks.

The client can be an Internet browser. Most of the client functionality was implemented using HTML and JavaScript. The graphical editing was implemented using a Java applet.

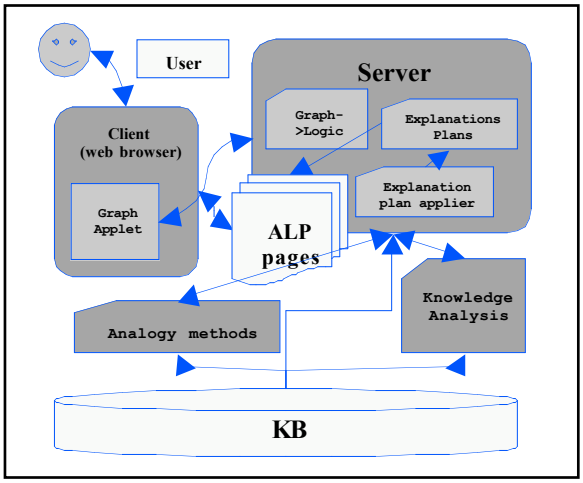


Figure1: SHAKEN architecture

To facilitate writing and debugging functions that generate HTML pages, we implemented a mechanism called Active Lisp Pages (ALP) (Rodriguez 2000), analogous to Microsoft's ASP (Active Server pages). This mechanism allows the developers to mix HTML code and Lisp code.

On the client side, most of the user interaction takes place in the graph-editing tool, and the Graph→Logic module ensures the translation between graphs and the KB representation.

Browsing a Knowledge Base

From the user's point of view, there are two sides to browsing a KB: finding concepts and understanding the content of those concepts.

Finding Concepts

SHAKEN provides two tools for finding concepts in a KB: a taxonomy browser and a search facility.

The taxonomic view of a KB presents the class taxonomy, which can be incrementally expanded and contracted at the request of a user.

Using the search facility, the user can type a partial character string, and the system will return concept names that match that string.

SHAKEN also provides semantic searching—a function that traverses the WordNet (Al-Halimi, Berwick et al. 1998) hierarchy for terms semantically related to concepts in the KB. As part of the documentation for each concept in our core KB, we have identified the WordNet entries that most closely match the semantics of the concept. SHAKEN's WordNet-based search tool finds the search term in WordNet, and then climbs the hierarchy of hypernyms (more general terms), finding all concepts listing those hypernyms in their documentation.

One of the advantages of semantic searching is that results are sorted according to the WordNet distance between the search term and the concept, and to the depth of the concept in the hierarchy. This gives preference to more specific concepts, meaning that the user is more likely to choose a more specific (and therefore more semantically loaded) concept than when browsing top-down through the tree.

If at any point during the user interaction, a user needs to select a concept, an entry point to the browsing facility is provided. To get more information about the concepts in the taxonomy or returned by search, the user can inspect the concept, either as a formatted description or as a graph.

Showing the Content of Concepts

The formatted concept description view presented here has two novel aspects:

- Well-organized display of information
- English-like text generation

Here, we describe these novel aspects in more detail.

Concept Description

The concept description view uses the notion of explanation design plans (EDPs) to organize the presentation of information. The explanation design plan idea comes originally from a work by James Lester (Lester and Porter 1997) aimed at generating automatic English explanation of the contents of a KB. An explanation plan encodes the contents and the organization of an explanation. Every explanation plan is a tree, and the nodes at a particular level in the tree represent topics. A node n_1 is a parent of a node n_2 if n_2 is a subtopic of n_1. Explanation plans employ three basic types of node:

- Exposition node: primary topic of an explanation
- Topic node: subtopic of an explanation
- Content specification node: specification of the content to include

Explanation plans are rooted at exposition nodes. An exposition node constitutes the highest-level grouping of content, and the children of exposition nodes are topic nodes. The children of topic nodes are content specification nodes, which name KB accessors. The child of a content specification node is a view, which is constructed by applying a KB accessor(Acker and Porter 1994). The realization system generates one or a small number of sentences for each view. In short, explanation plans store a formal representation of an explanation's content and organization.

Here are three examples corresponding to the three types of node used:

Exposition node: Process

```
Process Overview
   Text Description
   Type
   Participants
Process Details
   Qualitative Description
      Subevents
   Location Description
   Condition Description
   Temporal Information
   Object Function
      Telic Description
```

Topic node: Location description

```
Location Description
   Location
   Origination
   Destination
```

Content specification node: Location

```
Location
   Make-location-view
```

Each content specification node corresponds to a view in the KB: a view is simply a subset of slots of an object. For instance, the slot "is-between" is associated with the *Location* view. To display a view, we assemble the values of the slots associated with this view. In the above example, *make-location-view* is the name of the function that computes the values of this view.

Implementation of Explanation Design Plans. The EDPs are implemented in a single KB using an OKBC server. Each one of the nodes is represented by a frame in the KB. The KB contains two main EDPs: explain-process and explain-object. The hierarchical structure of the explanation plans is naturally represented as a hierarchy of frames.

For example, 'Exposition node' is represented as a class with two slots: 'Display name' and 'Children'. 'Display name' records the label used for displaying the heading of the information appearing under the exposition node, and 'Children' are the subheadings—for example, 'Qualitative description' and 'Location description'. 'Location description' is itself represented as a frame with two slots: 'Display name' and the name of the function used to compute the location information.

Displaying Knowledge Base Contents in English

Within the concept description defined above, the problem is to describe values of slots in the KB in a form that is as close as possible to the English an expert is used to, while not getting too far from the structure of the KB itself.

To achieve this objective, we have used a simple rule-based module that translates an expression from the KB formalism to an English sentence. This allows us to take into account both the general cases of common KB structures and some particular cases that otherwise would lead to clumsy sentences.

In full generality, displaying the contents of a KB in English would involve dealing with arbitrary logical sentences. As an initial simplification of this complexity, to display a concept, we create an example instance of it and compute various slot values by applying the relevant rules. As a result, we show only those slot values and never expose the actual rules to a user.

While some of the slot values are classes, many of them are Skolem individuals. Classes can be represented to a user simply by their names. But for Skolem individuals, we need a friendly mechanism for presentation. Since KM KBs also contain STRIPS representation of actions, we also need some way of displaying the add/delete precondition and negated condition lists. Here, we discuss how we generate English for each of those types of KB content.

Displaying Skolem Individuals. In the simplest case, a Skolem individual is displayed as "a <CLASS>" where <CLASS> is the direct type of that individual.

For individuals that are instances of abstract classes, it is not very friendly to display text such as "a Tangible-Entity" or "a Place". Therefore, we instead display how this Skolem individual is related to the concept that is being currently displayed. For example "a Place" may be replaced by "the location of the object of the Invasion". In other words, we

need to find a path of slots between the Skolem individual and the concept being displayed.

Here is a brief description of the algorithm we use to find the path. We start from the concept being displayed (the root) and loop over its children (the set of its slot values) and recurse until we hit the target.

```
Find-path(nd, tgt):
  if path{nd, tgt}
    then return it
    else
    for slot ∈ get-frame-slots(nd)
      ∃ find-path(get-slot-value(nd, slot), tgt)
    path{nd, tgt} ← slot & path-found
```

We also want to avoid paths that are overly complex and not easily understood. For example, "the location of the object of the next-event of the first-subevent of the Invasion" can naturally be further simplified. To keep the paths meaningful, we use the following two techniques:

- Look only for the shortest path.
- Stop searching for a path as soon as we encounter a class that is not too general.

These two heuristics proved sufficient in practice.

Displaying conditions: add- and delete-lists. SHAKEN uses a STRIPS-style representation for encoding change. Actions are events that change the state of the world. Thus, the application of an action in a situation is modeled by the creation of a new situation, reflecting the new world state after the action has been performed.

Actions are described using four lists, namely, the 'pcs_list' (preconditions list), the 'ncs_list' (negated preconditions list), the 'add_list' (add list), and the 'del_list' (delete list). The pcs_list (resp. ncs-list) contains a list of ground literals that are necessarily true (resp false) before an action is performed; the add_list (resp del-list) contains propositions that are necessarily true (resp false) after the action is performed. These lists are stored as slot values, on the frame representing the action. A proposition is a reified expression (i.e., an expression represented as an object), and allows us to make statements about that proposition P—for example, "Fred believes P"— or, for our purposes here, "the result of doing X is P"'.

A proposition is represented in KM by the structure `(:triple frame slot value)`, which denotes the assertion that `frame`'s `slot` includes `value`.

For instance, the class Move will have in its add list the triple `(:triple (the object of Self) location (the destination of Self))`, which means that when the action Move is simulated, a new situation is created where the value of the slot location for the object of the Move is the same as the value of the slot destination of this instance of Move.

To get the results shown on the second column, we used a recursive rule-based pattern matcher that translates KM expressions into lists of words.

Form	*Display*
(:triple (the object of Self) location (the destination of Self))	The location of the object of the action must be the destination of the action
(:triple _Breach12 result (a Be-Broken with (object Membrane13))	The result of the Breach is that the Membrane is broken

The second example illustrates a need to be able to generate meaningful text for specific objects, in this case a specific Be-Broken whose object is a particular Membrane. When a fuller description of such a concept is desired, SHAKEN can generate an English description that includes selected slots of a concept. The slots relevant in a concise description depend on which concept is being described.

Displaying Axioms. In SHAKEN's core KB we have encoded relevant English phrases and the grammar rules for combining them. The phrases and rules are distributed throughout the KB, allowing text generation particular to each concept. Rules and phrases inherit, but may be overridden by more specific concepts in the taxonomy. New concepts inherit default text generation rules.

Here are some examples of the description generated for particular concepts:

Form	*Display*
(a Move)	Something moves.
(a Move with (object _Car5) (agent *Wilma))	Wilma moves the car.
(a Be-Touching with (object _Wire6 _Terminal7)	The wire and the terminal are touching
(a Deliver with (object _Mail9) (recipient *Stacy))	The mail gets delivered to Stacy

Editing a Knowledge Base

A comprehensive solution for editing a KB should support the editing of several kinds of knowledge, such as class-subclass, slot values, constraints on slot values, axioms, and process knowledge. Our previous work on GKB-Editor was directed at editing class hierarchies, slots, slot values, and slot constraints. The focus in the present work is on editing axioms. The editing system described here enables the knowledge entry of an interesting class of axioms by abstraction from the example graphical description of a concept. The technical details of the abstraction process and mapping from directed graphs to logical form are described elsewhere (Clark, Thompson et al. 2001) and are not a primary subject of discussion here. Instead, we focus on the

graphical aspects of the tool, and assuming the availability of a module to convert graphs to logic, show how graphs can be used for entering add/delete lists, for analogical correspondences, and for asking questions.

From Graphs to Axioms

To use graphs for editing axioms, we need a scheme that, given an axiom, defines its graphical presentation and vice versa. To present the axioms about a concept, the raw axioms are not presented directly, but through an example of the concept, as a set of ground facts. Ground facts are graphable and provide a summary of the concept.

For instance, suppose the user wants to build a representation of how a virus invades a cell. To display this concept to the user, the system creates an instance of this class, and for each slot of these classes creates Skolem individuals corresponding to the value of that slot. This process is recursive, and is initially applied to a fixed depth limit. The user can later selectively expand portions of the graph.

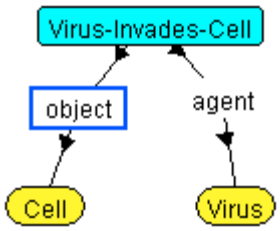


Figure 2: Early graph of VirusInvadesCell

Then, a graph is displayed, representing the concept VirusInvadesCell. Each node of the graph represents one of the Skolem individuals created. Each oriented link represents a slot, the end of the link being the value of the slot.

For such a graph, axioms could be synthesized as follows: first, the axiom is rephrased to mention only the "root" Skolem individual. For instance "Tangible-Entity3 is a Cell" becomes "the object of VirusInvadesCell1 is a Cell". This means that a path of relationships from the root to the instance replaces every "non root" individual. Then, it is generalized to hold for all instances of the Concept being defined. The final axiom thus has the form:

```
∀r isa (r , VirusInvadesCell)  ⇒
(∀v objectr(r, v) ⇒isa (v, Cell))
```

Look and Feel of the Interface. The graphical interface to manipulate graphs is flexible enough to allow the subject matter expert (or end user) to feel like he is using a drawing application, but at the same time to allow construction of only "meaningful" and "correct" graphs. A tool called Concept Maps (Novak 2001) has been built by research members of the team at the University of West Florida. The goal of Concept Maps is to allow people to represent, organize, and share knowledge, knowledge being described as relationships between concepts and being communicated in terms of graphs. The Concept Maps tool allows arbitrary graphs without any logical semantics. The graphical interface in SHAKEN is designed to feel as nonrestrictive as possible, while at the same time allowing only operations that have rigorous declarative semantics.

Since we are dealing with graphs, we have many helpful mechanisms other than written text with which to convey information to the user. A list of such mechanisms would include color and size of elements, distance between them, and horizontal/vertical order. The interface allows the user the use of such mechanisms to facilitate the construction of knowledge.

Operations to Manage the Graph. Most graphs representing concepts are virtually infinite, since for a KB to be useful, all the concepts are linked together in one sense or another. Therefore, we limit the graph we present to the user, but in the meantime, she has the ability to explore deeply into the structure of a concept. We achieve that by first limiting the depth of the initial display that is presented to the user to one or two levels, and second by managing two types of slot according to their importance. The initial display of a component shows only the abridged description. The user can then expand each one of the display nodes and also decide to view the full description. Nodes can be further expanded or contracted.

Layout Scheme. In essence, the display of graphs is hierarchical, since it starts from the concept being represented (shown in a different color), and then displays the values of its slots. Each of these values is itself a concept, so, it can be represented the same way. That does not mean that the graphs represented are trees or even directed acyclic graphs, since some of the nodes being expanded could point to nodes already present in the graphs.

The first time a concept map is drawn, the layout is automatically generated by the system, using a simple tree drawing algorithm. Assuming all the edges "cost" the same, the minimal spanning tree is calculated and the position for each node is determined. Then, all the edges from the original graph are added, completing the initial representation of the graph. Because the nodes are text, the original tree-drawing algorithm tends to generate excessively wide graphs. To resolve this, we stack nodes having the exact same edge to their parent. This approach shows trees that are more balanced in the vertical and horizontal directions.

Once the starting tree is drawn, the user is in complete control of how things will look. The user can move nodes around, organize them, and display or undisplay selectively. The user can drill down into the concepts, or add more nodes to the graph. The rendering engine must find space in the panel to fit in the nodes in a way that is intuitive and as unobtrusive as possible, so as to not interfere with the nodes that are already in the panel. This type of graph drawing algorithm is called "incremental layout algorithms". SHAKEN takes the following approach to an incremental layout: whenever it encounters a node that has not been

rendered, it finds the first ancestor that existed in the original graph, and uses the tree drawing algorithm locally, starting from that existing node. The local tree drawing must respect position for preexisting nodes; otherwise, a new node pointing back to some original node might reorganize the whole graph. This approach might overlap certain nodes, but under the presumption that the user will reorganize things as she sees fit, this is not a big issue.

Implementation of the Interface. The graphical tool is implemented as a Java applet that is embedded in an HTML page. The use of a Java applet is important because, since SHAKEN is a geographically distributed research application, we are able to deliver ongoing developments to the rest of the team and potential end users. The applet can then communicate to the server by using XML messaging over HTTP. The communication between the applet and its context page (the browser) is kept at a minimum.

The typical interaction between a user and SHAKEN's graphical toolkit is for the user to open a concept (for construction or browsing). The applet is given, as a parameter, the name and nature of that concept. Then, the conversation between the applet and the server begins: the applet sends a message requesting the concept, and the server responds by sending an XML representation of the graph (loosely based on GraphXML (Herman 2000)). The applet then must parse the XML message and take appropriate action. All communication between the applet and the server occurs in this fashion. To avoid a potentially big communication overhead, the applet accumulates state (i.e., position of nodes) about the graph at the client end. It communicates with the server only when absolutely necessary.

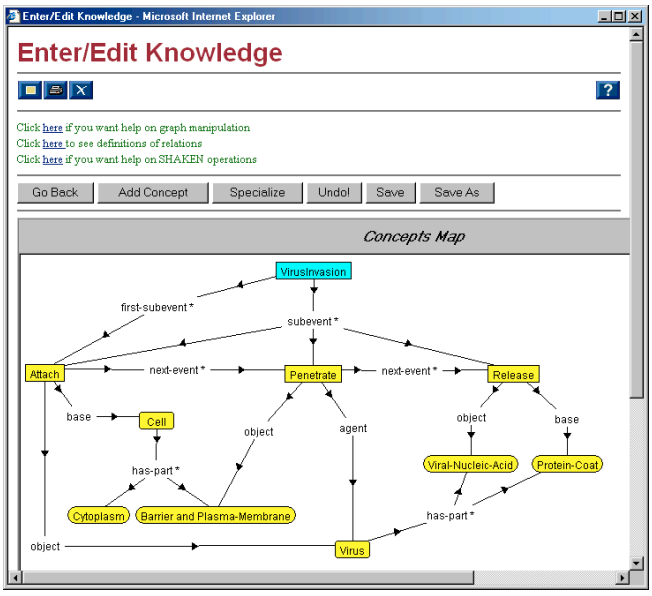


Figure 3: View of the graphical interface

Graphs for Knowledge Entry

Graphical editing operations were described in detail in a previous paper (Clark, Thompson et al. 2001). Here, we summarize those operations.

Add Concept. This operation consists in selecting any existing concept and adding it to the graph being edited. This does not link the user to any other node of the graph.

Specialize. This means selecting an existing node and changing its class by selecting one of its subclasses.

Connect. This is probably the most important operation: it allows the user to draw a relation between two concepts of the graph: she simply draws a line between two nodes, and then the system asks her to select a relation from a list of relations compatible with the two concepts being connected.

Unify. The user drags one of the nodes and drops it on top of another node. After confirmation, the two concepts are now considered to be the same object.

Querying a Knowledge Base

SHAKEN has a Question Answering facility that enables a user to ask questions about concepts in the KB. The question-answering interface is designed as a collection of fill-in-the-blank templates. Here are some example question templates:

- What is the <relation> of a <concept>?
- How does <event> occur in <concept> after <subevent>?

These questions may be instantiated by picking the values of the template variables. For example, the above two questions may be instantiated to

- What is the agent of Virus Invasion?
- How does a Copy occur in a DNA Transcription after a Move?

The user can choose the relation names from a menu, and choose the concept names by using the search facility. Very often, the variables in a parameterized question are not concept names, but logical expressions. For example, one may want to ask, "what is the agent of the Penetrate subevent of a Virus Invasion?" There is no stand-alone concept in the KB that represents the Penetrate subevent of Virus invasion. The graphical interface can be used to select such concepts. To do so, the user first selects a base concept, which in this case is Virus Invasion. He then opens the concept of Virus Invasion, and chooses the Penetrate subevent.

This choice returns to the question answering system the logical expression representing Penetrate that serves as the basis for question answering. The questions will then be titled: "Questions about a Penetrate as subevent of the Virus Invasion"

Expected Effects

The knowledge analysis module, called KANAL, allows a user to test the representation of a process (Kim and Gil 2001). KANAL functions by running an *animation* of the process and reports whether the preconditions of every step in the process hold true and also reports the changing slot values. The user can refine this testing by specifying which properties are expected to be true as a result of executing the process. For example, when testing a Virus Invasion process, a user may want to say that after the process is executed, the viral nucleic acid is inside the cell. A complete specification requires us to choose objects involved. To do so the user opens the graph representing the Virus Invasion, selects the Viral-Nucleic-Acid, then the Cell, and finally chooses a relation from a list compatible with the two objects selected.

Evaluation

During the summer of 2001, SHAKEN was extensively evaluated by IET (Information Extraction and Transport, Inc., www.iet.com). IET hired four biologists (three graduate students and one senior undergraduate) who had no background in computer science or knowledge representation. We trained the biologists to use SHAKEN, but training was limited to four days and our subsequent interactions with them were through an IET intermediary and restricted to fixing bugs in the system.

During the next four weeks, each biologist, working independently, was asked to build a KB to represent an n-page section of a college-level text on cell biology (Alberts, Bray et al. 1997). Along the way, IET asked each biologist to pose a set of questions to that biologist's own KB. The questions were drawn from standard test banks, and were presented in English. Each biologist "translated" the questions into SHAKEN's templates, and IET evaluated the responses.

Meanwhile, a pair of knowledge engineers, with significant training in knowledge representation and some background in biology, performed the same tasks by using SHAKEN. (These two people helped design and build SHAKEN.) This enabled IET to compare the biologists' KBs with those of knowledge engineers, as measured by the quality of the answers they produced.

This evaluation yielded a substantial body of data, which was then analyzed and summarized (Cohen, Chaudhri et al. 1999). The results that are most relevant to this paper examine the difference in answering ability between the KBs built by biologists and the ones built by knowledge engineers. Again, our goal is to develop tools, such as SHAKEN, that enable domain experts to build good KBs, whose quality is at least comparable to ones built by knowledge engineers.

IET hired another biologist to grade SHAKEN's answers to the questions posed by the KB builders. Each answer was assigned a score of 0 (lowest) to 3 (highest) on each of three criteria: answer correctness, quality of the representation, and quality of the explanation. These scores were averaged to give an *overall* score.

The mean overall score for answers generated from the biologists' KBs was 2.07, compared with 2.35 for knowledge engineers. This difference is small, but statistically significant ($F=69.32$, $p\sim0$). Moreover, there was considerable variance in overall scores across the set of six KBs:

KB creator	*overall score*
biologist 1	2.48
knowledge engineer 1	2.44
knowledge engineer 2	2.26
biologist 2	2.12
biologist 3	2.02
biologist 4	1.66

Table 1: Overall scores of knowledge bases built by biologists and knowledge engineers, sorted in descending order

The best-performing biologist was comparable with the best knowledge engineer, but the other three biologists performed less well than the second-ranked knowledge engineer. However, the overall result is most encouraging: SHAKEN can enable a domain expert to build a competent KB, and one that is comparable to one built by an experienced knowledge engineer.

Future Work

In the near future, we envision extending the editor in several directions: (1) entering analogical knowledge, (2) displaying changing fluents, (3) managing incremental layouts, and (4) showing axioms over and above a group of Skolem individuals.

Analogical Knowledge

Analogy is a powerful medium for communication used by humans. We plan to extend the current interface so that it could show two concept maps side by side and draw correspondences between the two. By using a partial specification of correspondences, the system would use the analogy engine to infer new relationships, thus greatly speeding the knowledge entry rate.

Fluents

While showing a process as a graph, the current interface shows properties of various concepts as of the beginning of the process. Currently, there is no way to show the

properties as they change with the execution of the process. We plan to extend the interface so that time changing properties could be shown.

Other Axioms

As we explained earlier, the only axioms that the user can graphically view and edit are the ones that can be translated into relationships between Skolem instances. Basically, most of these axioms are of the form

$\forall x \text{ isa}(x \text{ Class1}) \rightarrow \exists y \text{ isa } (y \text{ Class2}) \wedge \text{slot}(x\ y)$

or more generally axioms of the form

$\forall x \text{ isa}(x \text{ Class1}) \rightarrow \exists y \text{ isa } (y \text{ Class2}) \text{ p}(x\ y\ \ldots)$

Furthermore, the current system shows only binary relations. We plan to expand the class of axioms that can be graphically represented.

Conclusion

We have described a system that enables domain experts, unassisted by AI technologists to construct competent knowledge bases. We described in detail two components of this system that helped us achieve this goal.

The first component is a well-organized English-like presentation of KB content, which makes use of Explanation Design Plans, combined with an English generation tool built on a recursive rule-based pattern matcher. The second component is a graphical interface to enter knowledge and query the KB, extending the notion of Concept Maps.

We evaluated the system in a controlled experiment with domain experts, which suggests that these tools indeed enable domain experts to build KBs comparable to those built by knowledge engineers.

Acknowledgments

This work was supported by DARPA's Rapid Knowledge Formation project. We thank all the members of the SRI team who have helped implementing SHAKEN: Boeing, ISI, KSL Stanford, NWU, UT Austin, and UWF.

References

Acker, L. and B. Porter (1994). Extracting Viewpoints from Knowledge bases. National Conference on Artificial Intelligence.

Alberts, B., D. Bray, et al. (1997). Essential Cell Biology: An Introduction to the Molecular Biology of the Cell.

Al-Halimi, R., R. C. Berwick, et al. (1998). Wordnet, and Electronic Lexical Datbase, MIT Press.

Blythe, J., J. Kim, et al. (2001). An Integrated Environment for Knowledge Acquisition. International Conference on Intelligent User Interfaces.

Clark, P. and B. Porter (1999). KM The Knowledge Machine 1.4 - Users Manual. Austin, TX, University of Texas.

Clark, P., J. Thompson, et al. (2001). Knowledge Entry as the Graphical Assembly of Components. First International Conference on Knowledge Capture.

Cohen, P., V. K. Chaudhri, et al. (1999). Does Prior Knowledge Facilitate the Development of Knowledge-based Systems. National Conference on Artificial Intelligence.

Domingue, J., E. Motta, et al. (1999). Knowledge Modelling in WebOnto and OCML.

Herman, D. I. (2000). GraphXML - An XML Based Graph Interchange Format.

Kim, J. and Y. Gil (2001). Knowledge Analysis on Process Models. IJCAI.

Lester, J. and B. Porter (1997). Developing and Empirically Evaluating Robust Explanation Generators: The KNIGHT Experiments.

Maedche, A. (2000). Ontology Engineering Environment OntoEdit.

Mahalingam, K. and M. N. Huhns (1997). An Ontology Tool for Query Formulation in an Agent-Based Context. Center for Information Technology - Department of Electrical and Computer Engineering - University of South Carolina.

Novak, J. D. (2001). The Theory Underlying Concept Maps and How To Construct Them, University of West Florida.

Noy, N. F., M. Sintek, et al. (2001). Creating Semantic Web Contents with Protege-2000. IEEE Intelligent Systems.

Paley, S. and P. Karp (1996). GKB Editor User Manual, SR International.

Rodriguez, A. (2000). Active Lisp Pages (ALP), SRI International.

Swartout, B., R. Patil, et al. (1998). Ontosaurus: A Tool for Browsing and Editing Ontologies.