A Grammar-Driven Knowledge Acquisition Tool that incorporates Constraint Propagation

Simon White[*]
Accelrys Ltd.,
230/250 The Quorum,
Barnwell Road, Cambridge, CB5 8RE.
England, UK.
email swhite@accelrys.com

Derek Sleeman
Computing Science Department,
University of Aberdeen.
Old Aberdeen, AB24 3FX
Scotland, UK.
email dsleeman@csd.abdn.ac.uk

Abstract

To acquire knowledge that is fit for a specific purpose, it is very desirable to have a structured, declarative expression of the knowledge that is needed. This paper introduces a stand-alone knowledge acquisition tool, called COCKATOO (Constraint-Capable Knowledge Acquisition Tool), which uses constraint technology to specify the knowledge it requires. The language in which these specifications are given is based on the meta-language notation of context-free grammars. However, we also took the opportunity to build a tool that is both more flexible and powerful by augmenting context-free grammars with the expressiveness of constraints. COCKATOO was implemented using the SCREAMER+ declarative constraints package.

Keywords

Knowledge Acquisition, Formal Grammars, Constraints, Constraint-Augmented Grammars, SCREAMER+

INTRODUCTION

Previous work has addressed the problem of determining whether existing KBs (Knowledge Bases) can be used together with a selected problem-solver to satisfy a given problem-solving goal [15], [16]. We refer to this task as assessing the *fitness-for-purpose* of a KB. When the assessment identifies a mismatch between the given KBs and the problem-solver's expected KBs, we recognise two possible responses. Either the available KBs are totally inappropriate, in which case it is necessary to acquire them *ab initio*, or the existing KBs are close to being usable but need to be modified (possibly in a number of ways). This paper addresses the first of these actions; namely, to *acquire knowledge ab initio such that it meets the problem solver's requirements*. In current practice, a knowledge engineer uses a knowledge acquisition tool, or other elicitation method(s), to acquire the knowledge needed by a problem solver. Afterwards, the knowledge must usually be transformed, because the output of the knowledge acquisition tool cannot be used directly as input to the problem solver. We call such transformations *post-acquisitional transformations*. In this paper, we introduce the COCKATOO knowledge acquisition tool, which aims to minimise the need for post-acquisitional transformations. Our tool is *generic*, in the sense that it is independent of task, problem solver, and domain. On the other hand, it is highly configurable, and can be configured to acquire knowledge suited to a particular purpose (i.e., a problem-solving role).

The paper is organised as follows. In the next section, 'Grammar-Driven Knowledge Elicitation', we argue the benefits of using a context-free grammar as the basis for the specification of knowledge to satisfy a problem-solving role. In the section on 'Constraint-Augmented Grammars', we highlight some of the limitations of a purely grammar-driven approach to this task, suggesting the judicial use of constraints within the grammar to overcome some of the difficulties. We call this formalism a *constraint-augmented grammar*. The constraints are expressed using the declarative constraints package SCREAMER+ [14], [16], an extension of SCREAMER [11], [12]. The section on 'Grammar Development and Maintenance' outlines a process for building a constraint-augmented grammar that supports knowledge capture. Finally, we discuss the benefits and limitations of our work, and relate it to other work in the field.

GRAMMAR-DRIVEN KNOWLEDGE ELICITATION

When eliciting knowledge, it is desirable to have a structured, declarative specification of the body of knowledge that needs to be acquired. This can be used as both the target of the knowledge acquisition process and the criterion by which the acquired knowledge is assessed. Formal grammars provide a means for specifying knowledge to be acquired, are structured and declarative, and are also widely understood by knowledge engineers and computer scientists. However, there is an important difference in the way

K-CAP'01, October 22-23, 2001, Victoria, British Columbia, Canada.

[*] also affiliated with the Computing Science Department, University of Aberdeen, Old Aberdeen, AB24 3FX, Scotland, UK.

```
formation → <lithology>+
lithology → (<rock> <lithology-depth> [<lithology-length>])
rock      → (<rock-type> <rock-hardness>)
rock-type → (shale | clay | chalk | granite | other)
rock-hardness → (very-soft | soft | medium | hard | very-hard)
```

Figure 1: EBNF Grammar for acquiring knowledge of rock formations

```
(defclause formation ::= (repeat+ <lithology>))
(defclause lithology ::= (seq <rock> <lithology-depth> (optional <lithology-length>)))
(defclause rock      ::= (seq <rock-type> <rock-hardness>)
(defclause rock-type ::= (one-of 'shale 'clay 'chalk 'granite 'other))
(defclause rock-hardness ::= (one-of 'very-soft 'soft 'medium 'hard 'very-hard))
```

Figure 2: COCKATOO Grammar for acquiring knowledge of rock formations

that formal grammars are "traditionally" used, and the way that they have been applied here. Traditionally, grammars are used to solve the *parsing problem*; that is, to determine whether some *given* text conforms to some *given* formal grammar. For example, a C compiler must determine whether a given program consists entirely of legal C syntax. In grammar-driven knowledge elicitation, however, one attempts to *acquire* structured text *such that* it conforms to the given grammar.

We chose to represent EBNF grammars using a "LISPified" equivalent to the meta-notation of EBNF. This meta-language needs to:

- provide for the definition of grammar clauses of the target language;
- differentiate between a grammar's terminal and non-terminal symbols;
- provide the standard operators of an EBNF grammar, namely, *sequential composition*, the expression of *alternatives*, *repetition*, and *optionality*.

We illustrate our ideas with a simplified example from the domain of petroleum geology, and, in particular, the acquisition of a case base of oil well drilling experiences. The knowledge captured in this way is used to support subsequent drill bit run modelling and optimisation; for example, to help choose the right drill bit for a given formation sequence [8]. The EBNF grammar in figure 1 both describes and specifies a rock formation and its constituent lithologies (basic rock-types). The same grammar can be expressed in COCKATOO's syntax as shown in figure 2. (The correctness of the domain knowledge in our example has *not* been verified by a domain expert.)

Note that the non-terminal symbols `lithology-depth` and `lithology-length` have numeric values, and are more difficult to specify concisely with a grammar. We return to this issue in the following section. Note also that although our simple example illustrates only repetitions of 'one or more' (in this case, lithologies), COCKATOO also provides for repetitions of 'zero or more' with the keyword '`repeat*`'.

COCKATOO grammars are interpreted top-down, left to right. Usually, a special parameter to the `defgrammar` macro (not described here for lack of space) informs COCKATOO which is the 'top-most' grammar clause. So, for example, in the grammar of figure 2, we would tell COCKATOO to start with the `formation` clause. The interpretation of this clause leads to the acquisition of a repetition of lithologies, each in turn consisting of a sequence of a `rock`, a `lithology-depth`, and an optional `lithology-length`. A `rock`, in turn, consists of a sequence of a `rock-type` and a `rock-hardness`. The acquisition of either of these two non-terminal symbols involves the capture of a decision from the user among a number of distinct options (e.g., `shale`, `clay`, `chalk`, `granite` or `other`). These options are presented on-screen to the user by COCKATOO, so that a choice can be made and recorded. COCKATOO is sensitive to the number of possible values available. If there are too many values to be listed (i.e., more than a configurable upper limit), then the upper and lower bounds of the symbol (internally, a constraint variable) are provided to the user as additional support. If these values are not available at acquisition time, then the user is dependent upon the guidance provided by the knowledge engineer in the form of comments and questions (see below).

It is unrealistic to expect users to base their interaction with a knowledge acquisition tool on their understanding of an EBNF grammar. To help the user understand what information is required, and how it can be supplied, each clause of a grammar can be "decorated" with a question and/or a comment. A *question* should be a request for feedback which is directed at the user, such as "`What is the rock-type?`". A *comment* provides additional information, such as the meaning of particular terms, the exact format of the input, or other explanatory or "small-print" material. An example comment for the lithology clause might be

"`A lithology consists of a rock-type, a depth, an optional length, and a hardness`".

Even when the expert provides the knowledge acquisition tool with the *knowledge content* required by a problem solver, the format of the expert's inputs are seldom exactly the same as the format required by the problem solver. Usually, some kind of syntactic transformation needs to be performed. To achieve this functionality, COCKATOO allows a post-processing function to be specified for each clause. This is a single argument function that is applied to the value acquired by the clause. As a simple example, a question which the expert answered with 'yes' or 'no' is more likely to be represented by a LISP program with `t` (true) or `nil` (false). The post-processing function converts the terminology/representation of the expert to that of the problem solver. Note that the mechanism accommodates *arbitrary* post-acquisitional transformations. We have used this mechanism for simple *syntactic* transformations; we believe it could also be used as the call-out mechanism for supporting deeper *semantic* transformations. Currently, the full power of this mechanism is available only to knowledge engineers who are competent in LISP; later, we may devise a more user friendly interface for the description of such transformations. Additionally, we may allow adapters written in other languages to be linked in.

CONSTRAINT-AUGMENTED GRAMMARS

This section shows how a knowledge elicitation grammar can be augmented with constraint expressions. We claim that this can improve the conciseness and readability of the grammar, reduce its development time, and enhance its expressiveness. This view of knowledge elicitation is not inconsistent with the definition of a constraint satisfaction problem (CSP). (A CSP is defined by a set of *variables*, each of which has a known *domain*, and a set of *constraints* on some or all of these variables. The solution to a CSP is a set of *variable-value assignments* that satisfy all the constraints.) For example, consider a structured interview in which the answers to the knowledge engineer's agenda of questions are the variables of the problem, and there are concrete expectations about what their allowable values (the variables' domains) might be.

As we have seen, Grammar-Driven Knowledge Elicitation is a precise and powerful mechanism for acquiring knowledge. However, by combining the grammar-driven approach with constraint technology, we gain the following advantages.

Concise Specifications — Knowledge specifications for some tasks can be written much more concisely, thus giving a more readable specification, and also saving development time.

Single-Input Property Checking — The required properties of each user input can be checked at *acquisition time*, rather than prior to problem solving or at problem-solving time. That is, inadmissible values are identified early in the knowledge acquisition cycle. The properties that help to identify the admissibility of an input value are expressed naturally as constraints.

Multiple-Input Property Checking — Required properties of *multiple* inputs can also be checked at *acquisition time*. A property of this kind is expressed as a constraint among *multiple* inputs.

Reactive User-Interfaces — Constraints among multiple inputs can be constructed in such a way that the user-interface appears to react to the user's inputs. For example, the choice of a particular value for one input might narrow the domain of another.

Concise Specifications

The value of a COCKATOO clause can be specified by combining concrete values with the keywords `seq`, `one-of`, `optional`, `repeat+` and `repeat*`. Alternatively, a clause can be defined as an *arbitrary* LISP expression, such as a constraint expression. For example, the following concise clause accepts only an integer in the (inclusive) range 10 to 5000:

```
(defclause lithology-depth ::=
  (an-integer-betweenv 10 5000)
  :comment "The depth is given in metres (10 <=
depth <= 5000)")
```

With a purely grammar-driven approach, a part of the acquisition grammar would have to be dedicated to accepting either the sequence of characters that compose the integers of the range, or the enumeration of all acceptable values. For problems such as this, the simple constraint-based clause is much more maintainable than the equivalent grammar-based solutions without constraints.

Single-Input Property Checking

In the previous section, we argued that a grammar would be capable of describing the set of integers in the range 10-5000, but the introduction of constraints made the solution much more concise. For that problem, the constraint-based approach was no more *powerful* (in terms of expressiveness) than the purely grammar-based approach[1], though it clearly offered advantages. However, a constraint-augmented grammar also provides for the verification of properties beyond the power of a purely grammar-driven approach. As an example, consider prime numbers. It is not possible to define a formal grammar that admits any prime number, but disallows non-primes. However, a constraint-augmented grammar can include a clause that admits only prime numbers by constraining the input value to satisfy a predicate that tests for primeness[2].

---

[1] Both approaches solved the problem.

[2] The example is given in full in [16].

In LISP, membership of a type can be subject to satisfaction of a *arbitrary* LISP predicate, so the mechanism for checking the properties of a single input value is general and powerful.

Multiple-Input Property Checking

Another way of specifying values that could not be expressed by a context-free grammar is by asserting constraints across multiple input values. For example, a context-free grammar would not be able to constrain two variables to have different values unless it *explicitly* represented all those situations in which the values were different. At best, this represents much work for the implementer of the grammar. If the variables have an infinite domain, however, it is not even possible. The following clause returns a sequence of two rock-types which are constrained to be different.

```
(defclause rock-types ::=
  (let ((type-1 (make-variable))
        (type-2 (make-variable)))
    (assert! (not-equalv type-1 type-2))
    (seq type-1 type-2)))
```

A similar technique can be used in the grammar given earlier to prevent the rock-types of consecutively acquired lithologies to be the same (if consecutive rock-types were the same, it would be a single lithology). When acquiring a value for this clause, the second value must be different to the first:

```
LISP> (acquire (find-clause 'rock-types))
Input a value: granite
Input a value: granite
That value causes a conflict. Please try another
value...

Input a value: shale
(GRANITE SHALE)
```

Here, `(GRANITE SHALE)`, is the return value of the `rock-types` clause.

Reactive User-Interfaces

Constraints can also be used to modify the behaviour of the acquisition tool, depending on the values supplied by the expert. The idea is that inputting a value in answer to one question may trigger a reduction in the set of possible answers to a different question. This is the issue of *reactive knowledge acquisition* mentioned earlier. Reconsider the example of acquiring a pair of rock types that are constrained to be different. This time, we reuse the clause first given in figure 2 that acquires a single rock type:

```
(defclause rock-type ::=
  (one-of 'shale 'clay 'chalk 'granite 'other))
```

When this clause is interpreted, it creates a constraint variable whose domain (set of possible values) consists of the rock types `shale`, `clay`, `chalk`, `granite` and `other`. COCKATOO uses the domain of a variable when displaying the possible input values to the user. The following clause uses the `rock-type` clause to return a sequence of two rock types. The values of the rock types `type-1` and `type-2`, which are not known until acquisition time, are constrained to be different:

```
(defclause rock-types ::=
  (let ((type-1 (find-clause 'rock-type))
        (type-2 (find-clause 'rock-type)))
    (assert! (not-equalv (acquired-valuev type-1)
                         (acquired-valuev type-2)))
    (seq type-1 type-2)))
```

Note that the constraint on `type-1` and `type-2`, imposed by the `assert!` function, is declarative and therefore symmetrical, allowing either of the two values to be acquired first. The return value, on the other hand, is a sequence that is interpreted such that `type-1` is acquired before `type-2`. If the expert inputs `shale` as the first value of the pair, it should not be offered as a possible value for the second value of the pair. Instead, the user-interface should react to the expert's inputs, as illustrated below:

```
LISP> (acquire (find-clause 'rock-types))

The possible values are:
  A.  SHALE
  B.  CLAY
  C.  CHALK
  D.  GRANITE
  E.  OTHER
Which value? : granite

The possible values are:
  A.  SHALE
  B.  CLAY
  C.  CHALK
  D.  OTHER
Which value? : shale
(GRANITE SHALE)
```

When acquiring the second rock-type, `GRANITE` was not offered as a possible value because choosing that value would be inconsistent with the disequality constraint. Such behaviour cannot be realised by a context-free grammar because the rock-type is not known until acquisition time. A context-free grammar cannot build in such conditions at 'compile time'.

We have shown that the determination of a variable's value at acquisition time can cause the domain of another variable to be reduced. Sometimes, the domain of a variable might be reduced to a single value, causing that variable to become bound. When this happens, the value of that variable need not be acquired from the expert, as it has already been inferred.

GRAMMAR DEVELOPMENT AND MAINTENANCE

It is important for a knowledge specification to be easily readable so that persons other than the KA tool developer can understand it. Readable specifications tend to be easier to write, discuss, and maintain. COCKATOO has two main features that enhance both the readability and maintainability of its knowledge specifications. Firstly, it has developed an approach based on the use of (EBNF-like) formal grammars, which are a well-known type of formal specification, and in widespread use. Secondly, COCKATOO makes a clear separation between the knowledge specification and the acquisition engine that acquires the knowledge. This leads to concise specifications, which contain only pertinent material, as well as a general purpose acquisition tool which is reusable across domains. By way of contrast, consider a custom-tailored acquisition tool that embeds the knowledge specification within its program's source code. Such a tool would not be reusable across different problem domains, and even with optimal coding style, only those with knowledge of the programming language would be able to understand the specification.

COCKATOO already provides mechanisms for specifying the required knowledge at a "high level". That is, during (grammar) development, the knowledge engineer can concentrate on the nature of the knowledge to be acquired, rather than the program that acquires it. Grammar development using COCKATOO is a (cyclic) refinement process, which includes the following chronological stages.

Knowledge Analysis — The aim of this stage is to capture the most important concepts of the domain and the relationships between their instances. In effect, we aim to derive a basic domain ontology.

Grammar Construction — In this stage, we decide which of the ontological elements from the previous stage will be included in the knowledge capture. A further analysis of these elements (for example, a structural decomposition) leads to a grammar that captures the basic knowledge requirements in terms of those elements and their multiplicities (e.g., one-one, one-many, many-many).

Adding Constraints — An optional stage to enhance the grammar with *constraints*. When used, the aim of the stage is twofold: firstly, to remove unwanted or nonsensical input combinations from the specification; and secondly, to eliminate redundant questions.

Embellishment — Embellishing the grammar with questions and comments.

Notice that the system's communication with the expert is not considered until the final stage of development, reflecting the attention paid to the correctness of the knowledge specification in the early stages.

The examples presented throughout this paper have demonstrated COCKATOO's flexibility for use in many different domains. COCKATOO can also be configured *quickly* for use in a new domain. For example, three sample grammars presented in [16] were developed (and refined) in less than a day each! The main reason for the ease with which COCKATOO can be reused is the clear separation between the data that drives knowledge acquisition (the grammar) and the more generic tool that processes the data (the COCKATOO acquisition engine). COCKATOO is, in effect, a knowledge acquisition *shell* that supports the building of custom-tailored KA tools.

Although it was developed to acquire *knowledge bases* for use within the MUSKRAT toolbox [16], a KA tool such as COCKATOO has the potential to be applied to a very wide range of application domains. Not only can it acquire simple knowledge elements, it can manage complex constraint relationships between them, and post-process user inputs for further compatibility with other tools. With regard to COCKATOO's suitability for different acquisition tasks, it could be used in most situations that involve a substantial amount of numerical, textual or symbolic user input. It is well-suited to supporting knowledge acquisition for both classification and configuration (or limited design) tasks. For classification tasks, we must acquire example cases and their associated class; for configuration tasks, the building blocks of the design are well-known, but their combinations may be explored. Although all the design decisions are made by the human user, the output is nevertheless constrained to be within the "space" specified by the grammar.

DISCUSSION

This paper has argued the value of a declarative specification of the knowledge to be acquired, and introduced the COCKATOO tool, which acquires knowledge by interpreting a constraint-augmented grammar. This approach offers enhanced readability, eased maintenance, and a reduced initial development effort compared with the construction of multiple customised tools for different domains. Augmenting a context-free grammar with constraints increases both the expressiveness and conciseness of the notation. The power of the tool that interprets the notation is also increased because in some situations its behaviour can be altered by the user's responses to questions. Conciseness of the notation is improved because admissible values do not always have to be detailed down to the level of individual characters or symbols.

Related Work

COCKATOO is an automated knowledge elicitation tool, but differs considerably from current knowledge elicitation tools based on repertory grids, sorting, and laddering (see, for example, [1], [10], and PC-PACK[3]), because they cannot be tightly coupled with an application. The knowledge acquired using these tools must be post-processed "by hand" before they can be used by a problem solver or other

[3] PC-PACK is a software package marketed by Epistemics Ltd. See `www.epistemics.co.uk`

application program. COCKATOO, on the other hand, provides a very general post-processing facility which allows the acquired knowledge to be packaged in a form suitable for subsequent use.

Generalised Directive Models (GDMs) [13], [7] also use grammars, but for a different purpose to COCKATOO. COCKATOO uses a grammar to guide the acquisition of *domain knowledge* from a *domain expert*, such that the knowledge can be used by an *existing* problem solver. GDMs, on the other hand, apply grammars to assist *knowledge engineers* with *task decomposition* when *building* a knowledge-based system. The purpose of a GDM grammar is to guide the knowledge engineer to classify the task(s) at hand, so that an appropriate knowledge elicitation tool can be selected. Thus, the grammars of the two approaches describe sentences of quite different natures: a COCKATOO sentence describes a domain structure, whereas a GDM sentence describes the decomposition of a task into subtasks and their respective types. It may also help to consider the meanings of the terminal symbols in each of the two formalisms. A terminal symbol of a COCKATOO grammar represents a domain concept or value; a terminal symbol of a GDM represents a task type whose association with a knowledge elicitation tool is known. The two approaches are complementary, and could even be used together – a terminal node of the GDM grammar could be associated with COCKATOO as the most appropriate elicitation tool for the node's task type.

It is interesting to compare COCKATOO with the Protégé project and, in particular, the Maître tool [2]. Protégé, like COCKATOO, is a general tool (a "knowledge acquisition shell") whose output is in a format that can be read by other programs (CLIPS expert systems). Also, before using Protégé to acquire knowledge, the knowledge engineer must first configure it with an "Ontology Editor" subsystem, called Maître. This tool enables the knowledge engineer to define an ontology, which is then used as the basis for knowledge acquisition. The ontology plays the same role in Protégé as the constraint-augmented grammar in COCKATOO — it specifies the knowledge to be acquired. Another subsystem of Protégé, called Dash, can be used interactively to define a graphical user-interface for the KA tool. Although the graphical user interfaces are very appealing, we do not believe that Protégé has the same knowledge specification power as COCKATOO, since it is not supported by an underlying constraints engine.

The idea of minimising the number of questions asked of the user was inspired by the questioning techniques of MOLE [3], and the intelligent mode of the MLT Consultant [5]. However, the approach taken by COCKATOO is different from both of these systems. MOLE reduces the amount of questioning by making intelligent guesses about the values of undetermined variables, and subsequently requesting the user's feedback. MLT Consultant uses an information theoretic measure to determine which questions are asked. In contrast, COCKATOO uses local propagation techniques to identify redundant questions.

A so-called *Adaptive Form* [4] is a graphical user-interface for acquiring structured data that modifies its appearance depending on the user's inputs. For example, a form for entering personal details would only show a field for entering the spouse's name if the user had entered *married* in the *marital status* field. Although this kind of behaviour is very similar to the reactive knowledge acquisition of COCKATOO, Adaptive Forms are driven by (context-free and regular-expression) grammars alone, and do not support more complex constraints. The system uses *look-ahead parameters* to decide which unbound fields to display at any given time. We also note that Frank and Szekely extended their grammar notation to include 'labels', which have the same function as the questions of COCKATOO. They do not provide the equivalent of comments, name spaces, or post-processing. The Amulet system [6] does use a constraint solver to manage its user-interface, but employs it to control the positioning and interactions of graphical objects, rather than to support knowledge acquisition.

Limitations

COCKATOO currently does not allow the user to backtrack from a given user input, and try something else. Once an input has been entered, the user is committed to that value and cannot change it later. This is a serious limitation because not only does it not allow for typographical errors; it also prevents the user from experimenting with the COCKATOO grammar in a 'what-if' mode. Of course, COCKATOO can always be aborted and the acquisition restarted from the beginning, but a more flexible backtracking capability is a desirable feature that should be addressed. Ideally, at each stage of the acquisition process the user should be given the option to go back to the previous stage, retract the previous input, and input a new value. The problem is that retracting an input is not simple, because the constraint-based assertions associated with that input may already have caused propagation to other constraint variables. To retract an input, one must be able to recover the states of *all* constraint variables before the assertion was made. One option to achieve this functionality is to record the states of all constraint variables before each user input; another option is to record the changes that occur after each user input, so that they may be reversed. Ideally, the constraints package would provide a retraction facility of its own [16].

There are two types of propagation that should be supported by a KA tool for MUSKRAT; namely *inter-KB propagation* and *intra-KB propagation*. In the former case, knowledge is propagated from one knowledge base to a different knowledge base. In the latter case, knowledge is propagated from one part of a knowledge base to a different part of the *same* knowledge base.

For inter-KB propagation, we require a mechanism by which the knowledge contained in an *existing* knowledge base is made available to COCKATOO for acquiring a different, but related, knowledge base. We have not yet addressed this problem. However, COCKATOO already provides a mechanism for intra-KB propagation through the functions `find-clause` and `acquired-valuev`. The function `find-clause` can be used from within a grammar to search for a known clause, thus offering a facility for grammar introspection. The function `acquired-valuev` is used to make assertions about the value that a clause (eventually) acquires. These two functions can therefore be used together to specify at acquisition time the knowledge that some other clause should acquire. This is the behaviour of intra-KB propagation.

We feel that the usability of COCKATOO would be significantly improved by the addition of graphical user-interfaces at two levels. Firstly, a graphical user-interface 'front-end' to COCKATOO would provide the opportunity for enhanced end-user support; for example, through the use of distinct graphical input forms (with appropriate widgets, such as text boxes and drop-down menus). Secondly, a graphical user-interface could provide useful support for the acquisition of grammars, so that the knowledge engineer would no longer have to input COCKATOO grammars in their 'LISPified' syntax. Such a 'meta-tool' would output a COCKATOO grammar (perhaps as the result of post-processing). It would be interesting to investigate whether COCKATOO is flexible enough to act as both the meta-tool and the domain expert's tool.

COCKATOO will be used by a class of undergraduate students in the autumn of 2001, and we expect it to be used subsequently by the Advanced Knowledge Technologies (AKT) project.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support provided for this work through an EPSRC studentship.

REFERENCES

[1] Diaper, D., (1989), "Knowledge Elicitation: Principles, Techniques and Applications", Ellis Horwood, Chichester, England, UK.

[2] Eriksson, H., Puerta, A. R., Gennari, J. H., Rothenfluh, T. E., Samson, W. T., Musen, M., (1994), "Custom-Tailored Development Tools for Knowledge-Based Systems", Technical Report KSL-94-67, Section on Medical Informatics, Knowledge Systems Laboratory, Stanford University, California, USA.

[3] Eshelman, L., (1988), "MOLE: A Knowledge-Acquisition Tool for Cover-and-Differentiate Systems", in Marcus, S., (Ed.), "Automating Knowledge Acquisition for Expert Systems", Kluwer Academic Publishers, pp. 37-80.

[4] Frank, M. R., Szekely, P., (1998), "Adaptive Forms: An Interaction Technique for Entering Structured Data", Knowledge-Based Systems, Vol. 11, pp. 37-45.

[5] Kodratoff, Y., Sleeman, D., Uszynski, M., Causse, K., Craw, S., (1992), "Building a Machine Learning Toolbox", in Enhancing the Knowledge Engineering Process, Steels, L., Lepape, B., (Eds.), North-Holland, Elsevier Science Publishers, pp. 81-108.

[6] Myers, B. A., Mcdaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., And Doane, P., (1997), "The Amulet Environment: New Models for Effective User Interface Software Development", IEEE Transactions on Software Engineering, Vol. 23, No. 6, pp. 347-365.

[7] O'Hara, K., Shadbolt, N., Van Heijst, (1998), "Generalised Directive Models: Integrating Model Development and Knowledge Acquisition", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 497-522.

[8] Preece, A., Flett, A., Sleeman, D., Curry, D., Meany, N., Perry, P., (2001), "Better Knowledge Management through Knowledge Engineering", IEEE Intelligent Systems, Vol. 16, No. 1, pp. 36-43.

[9] Reichgelt, H., Shadbolt, N., (1992), "ProtoKEW: A knowledge-based system for knowledge acquisition", in Artificial Intelligence, Sleeman, D, and Bernsen, NO (Eds.), Research Directions in Cognitive Science: European Perspectives, volume 6, Lawrence Erlbaum, Hove, UK.

[10] Shadbolt, N. R., (2000), Knowledge Elicitation Techniques In Knowledge Engineering & Management: The CommonKADS Methodology. G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. van der Velde & B. Wielinga. Pub: MIT Press.

[11] Siskind, J. M., McAllester, D. A., (1993), "Nondeterministic LISP as a Substrate for Constraint Logic Programming", in proceedings of AAAI-93.

[12] Siskind, J. M., McAllester, D. A., (1994), "SCREAMER: A Portable Efficient Implementation of Nondeterministic Common LISP'", Technical Report IRCS-93-03, Uni. of Pennsylvania Inst. for Research in Cognitive Science.

[13] Van Heijst, G., Terpstra, P., Wielinga, B., Shadbolt, N., (1992), "Using generalised directive models in knowledge acquisition", in Proceedings of EKAW–92, Springer Verlag.

[14] White, S., Sleeman, D., (1998), "Constraint Handling in Common LISP", Department of Computing Science Technical Report AUCS/TR9805, University of Aberdeen, Aberdeen, UK.

[15] White, S., Sleeman, D., (1999), "A Constraint-Based Approach to the Description of Competence", in

Fensel, D., Studer, R., (Eds.), Proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modelling, and Management (EKAW–99), LNCS, Springer Verlag, pp. 291-308.

[16] White, S., (2000), "Enhancing Knowledge Acquisition with Constraint Technology", PhD Thesis, University of Aberdeen, Scotland, UK.