AUCS/TR0202

Uses of a Grammar-Driven Case-Acquisition Tool

Derek Sleeman[1] & Simon White[1, 2]
[1]Computing Science Department, University of Aberdeen,
Aberdeen, AB24 3FX Scotland, UK
email: dsleeman@csd.abdn.ac.uk
[2]Accelrys Ltd., 230/250 The Quorum, Barnwell Road,
Cambridge, CB5 8RE, England UK
email: swhite@accelrys.com

Abstract. This paper introduces a stand-alone case/knowledge acquisition tool, called COCKATOO (Constraint-Capable Knowledge Acquisition Tool), which uses an Extended BNF grammar to represent the main characteristics of the (domain) cases. Further, we also took the opportunity to build a tool that is both more flexible and powerful by augmenting the context-free grammars with the expressiveness of constraints. COCKATOO was implemented using the SCREAMER+ declarative constraints package. Additionally, the paper discusses several uses of the tool by both the developers and, more significantly, by a group of knowledge engineers.

Keywords
Case/Knowledge Acquisition, Formal Grammars, Constraints, Constraint-Augmented Grammars, SCREAMER+

Uses of a Grammar-Driven Case-Acquisition Tool

Derek Sleeman[1] & Simon White[1, 2]

[1]Computing Science Department, University of Aberdeen,
Aberdeen, AB24 3FX Scotland, UK
email: dsleeman@csd.abdn.ac.uk
[2]Accelrys Ltd., 230/250 The Quorum, Barnwell Road,
Cambridge, CB5 8RE, England UK
email: swhite@accelrys.com

Abstract. This paper introduces a stand-alone case/knowledge acquisition tool, called COCKATOO (Constraint-Capable Knowledge Acquisition Tool), which uses an Extended BNF grammar to represent the main characteristics of the (domain) cases. Further, we also took the opportunity to build a tool that is both more flexible and powerful by augmenting the context-free grammars with the expressiveness of constraints. COCKATOO was implemented using the SCREAMER+ declarative constraints package. Additionally, the paper discusses several uses of the tool by both the developers and, more significantly, by a group of knowledge engineers.

Keywords
Case/Knowledge Acquisition, Formal Grammars, Constraints, Constraint-Augmented Grammars, SCREAMER+

1 INTRODUCTION

We have reported elsewhere the problem of determining whether existing KBs (Knowledge Bases) can be used with a selected problem-solver to satisfy a given problem-solving goal [11], [12]. We have referred to this activity as assessing the *fitness-for-purpose* of a KB. When a mismatch is detected between the given KBs and the problem-solver's expected KBs, we recognise two possible responses. Either, the existing KBs are close to being usable but need to be modified (possibly in a number of ways), or the available KBs are totally inappropriate, in which case it is necessary to acquire them *ab initio*. This paper addresses the later situation; 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 is not directly usable as input to the problem solver. Such transformations are usually called *post-acquisitional transformations*. In this paper, we discuss the COCKATOO knowledge acquisition tool, which aims to minimise the need for post-acquisitional transformations. This tool is *generic*, in the sense that it is independent of task, problem solver, and domain; further, it is highly configurable.

In the second 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. The section on 'Constraint-Augmented Grammars' highlights 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 the resulting formalism a *constraint-augmented grammar*. The constraints are expressed using the declarative constraints package SCREAMER+ [10], [12], an extension of SCREAMER [8], [9]. The section on 'Grammar Development and Maintenance' outlines a procedure for developing a constraint-augmented grammar that supports knowledge capture. The fifth section discusses the use of the tool by a group of Knowledge Engineers. Finally, in section 6, we relate this work to other work in the field, and discuss its benefits and limitations.

2 GRAMMAR-DRIVEN KNOWLEDGE ELICITATION[1]

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, which are structured and declarative, and are also widely understood by knowledge engineers and computer scientists. However, there is an important difference in the way that formal grammars are "traditionally" used, and the way that they have been applied here.

```
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

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;

[1] With some minor adaptations, sections 2, 3 & 4 of this paper are taken from the corresponding sections of [13].

- 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 the modelling and optimisation of drill bit selection; for example, to help choose the right drill bit for a given formation sequence[6]. 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 informs COCKATOO which is the 'topmost' grammar clause, [12]. 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 onscreen 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.

3 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.

3.1. 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.

3.2 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[2], 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[3].

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.

3.3 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-

[2] Both approaches solved the problem.

[3] The example is given in full in [12].

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.

3.4 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 **granite** 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? : D

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

4 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 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 [12] 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 [12], 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.

5 EXAMPLES of USE

a) An example of a simple COCKATOO Script.

Appendix 1 gives the complete listing of an operational acquisitional script (well2) for the wells domain, and Appendix 2 gives an example interaction based on that. This shows the user inputting information about a well, which involves 4 strata and their depths. In fact, had the user input rocks which were identical in adjacent strata this annotation script would not have spotted it as the appropriate constraints have not been included. This is clearly desirable as 2 strata which are the same would, in practice, be handled as a single strata. How such checks can be achieved by constraints is discussed in the next sub-section.

Use of more complex constraints in COCKATOO Scripts Above we saw the effects of constraints on the options for strata which are presented to the user. Below we give an example of a simple constraint which asserts that:

strata 1 $\neq$ strata 2 & strata 2 $\neq$ strata 3

```
(defclause cl-rock-3 ::=
   (let ( (strata1 (make-variable)) (strata2 (make-
variable))
   (strata3 (make-variable)))
           (assert! (not-equalv strata1 strata2))
           (assert! (not-equalv strata2 strata3))
           (seq strata1 strata2 strata3)   ))
```

Note that the use of this clause is limited as it only allows the expert to specify strata in groups of three. A more general formulation, which allows the user to input any number of strata, and prohibits adjacent strata from being the same, is given by:

```
 (defclause well-strata ->
   (do* (
 (all-strata nil)
         stratum
     (previous-stratum (make-clause (one-of 'dummy-
 initial-value)))
         (continue? 'yes)  )
        ((eq continue? 'no) all-strata)
       (setq stratum (find-clause 'rock3))
       (assert! (not-equalv (acquired-valuev previous-
stratum)
(acquired-valuev stratum)))
       (acquire stratum)
       (setq all-strata (append all-strata (list stra-
tum)))
       (setq previous-stratum stratum)
       (setq continue? (acquire (make-clause (one-of
'yes 'no)
 :question "Another stratum?"))) ))
```

This clause is beginning to resemble plain LISP programming because it explicitly uses a LISP `do` loop. It is not an elegant solution, and has lost the simplicity of the EBNF grammars and simple constraints. Perhaps more significantly, the clause wrests control from the COCKATOO acquisition engine by calling `acquire` directly. Normally, `acquire` would only be invoked by the acquisition engine whenever variables need to be instantiated. Therefore this formulation is not really consistent with the COCKATOO approach. An alternative formulation, which is more consistent with the use of declarative constraints within grammars, is given below:

```
(defclause well-strata ->
  (let (
        (a (make-variable))
        (b (find-clause 'well-strata))
        )
    (assert! (not-equalv a (firstv (acquired-valuev
b))))
    (seq a (optional b :question "Another? ")) ))
```

This clause maintains two local variables: a constraint variable a, which is created and used to store a single acquired value, and an instance of a clause object, `b`, which is actually a recursive reference to the same clause. The clause asserts that the value acquired for the constraint variable `a` at acquisition time must be different to the first of the rest of the values acquired by the recursive application of the `well-strata` clause. (This assertion captures the constraint that consecutive strata be different.) The return value of the clause is a sequence, composed from the constraint variable a and an optional recursion on `well-strata`. Note that at the time of execution of the clause by LISP, the returned values are constraint variables. The constraint variables are subsequently instantiated by the COCKATOO acquisition engine. The above formulation looks elegant, however, it is not easy to understand because it is recursive & constraint-based; further, you need to distinguish between LISP's run-time and COCKATOO's acquisition-time phases.

b) **Usage of COCKATOO by a group of Knowledge Engineers**

Context. A final year Undergraduate class was given the following assignment:

" Firstly choose a non-Computer Science, preferably non-Technical, topic which interests you; for example: Cooking, Gardening, Diving....

Secondly, find a self contained article of about 10-15 pages (i.e. around 5000 words); for reasons which will be clear shortly it will be advantageous if this is available on-line. You should then carry out the following analyses on this text:

- Produce a detailed glossary of the unusual/domain-specific terms it contains.
- Prepare detailed notes for an interview which you might hold on this topic with a domain expert.
- Use an interactive MARKUP system to detect/indicate domain specific concepts, attributes and values, and with these interactively build an ontology.
- Choose an aspect of the topic for which you'd like to have a series of examples, create the appropriate COCKATOO script, and use the COCKATOO system to create the corresponding example set."

Topics selected by the Group Below we give a list of the topics selected by the 24 members of the class:

[Malayan] Dreams	Kayaking
Buddist Theology	Musical Instruments
Classification of Ants	Natural Magic
Coffee Consumption	Norse Mythology
Competition Shooting	Planets & Comets
Composition of planets	Plant Characteristics
Egyptian Mythology	Powered Paragliding
Firearms Safety	Canoeing Equipment
Gang Warfare	Selecting/Selling Snowboards
Greek Gods	Selection of Skis
Health Problems of Drug Addicts	Spacecraft
Jazz	UK Cities

Dialogues produced with these COCKATOO scripts
Appendix 3 shows a further interaction with COCKATOO based on the *Instrument Description* script produced by one of the group.

The Balance of the several scripts produced
As noted before there were 24 students in the group; one did not produce a COCKATOO annotation script. Of the 23 scripts produced only 3 contained constraints and these were simple constraints to ensure that numerical values fell within a pre-specified numerical range. A further 2 students used constraints to check that input was a string – that is they used this mechanism to perform type checking. So little use (12.5%) was made of COCKATOO's constraint mechanisms. This was possibly a consequence of the examples presented to the class. The principal example showed the use of a EBNF grammar whereas constraints were only introduced as a subsidiary topic with only "snippets" of scripts used to illustrate

them. In future, we plan to feature constraints more centrally in the examples presented to the group.

6 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.

6.1 Related Work

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.

A so-called *Adaptive Form* [3] 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 [5] 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 acquis ition.

For a more extensive discussion of future work please see [12] or [13].

6.2 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. Ideally, the constraints package would provide a retraction facility of its own [12].

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.

In section 5a) we discussed the use of constraints applied to a variable and perhaps unknown number of objects (in this case rock strata). In this application, we have encountered 3 types of constructs: BNF grammar statements, constraints, and iterative constructs. So far we have not been able to integrate these very satisfactorily in COCKATOO. It is likely that to make each of these constructs more transparent to the domain expert, we will have to represent them in a more intuitive way borrowing ideas from graphical programming [1]. Additionally, there is a need to enhance the debugging facilities available. Suppose one or more constraints are not satisfied at data acquisition time. Currently the User is told when a violation occurs but it might well be a number of unsatisfied constraints which have caused the problem and not just the last one. Again some way of making this process more transparent to the User is needed. (Note this is a general problem for CSP systems, [4].)

7 ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support provided for this work through an EPSRC studentship (SW). We are also grateful to the members of the Knowledge Technologies class, Computing Science Department, University of Aberdeen 2001/2 for using the COCKATOO system and providing us with valuable feedback. Appendix 3 is based on a script produced by Tamara Buchan.

8 REFERENCES

[1] diSessa, A. A. (1997) "Twenty reasons why you should use Boxer (instead of LOGO)". In M Turcsanyi-Szabo (Ed). *Learning & Exploring with Logo*: Proceedings of the Sixth European Logo Conference. Budapest Hungary, pp7-27.

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

[4] Freuder, E. C. (1997). "In Pursuit of the Holy Grail", *Constraints Journal*, Vol 2,1, pp57 - 61.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

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

[13] White, S., & Sleeman, D. (2001) “A Grammar-driven Knowledge Acquisition Tool that incorporates Constraint Propagation." KCAP-01 Conference, Victoria, Canada (October 2001), ACM Press, pp. 187-193.

APPENDIX 1.

The Wells COCKATOO ACQUISITION SCRIPT (slightly simplified)

```
;;; Start of Grammar Definition
(defgrammar drilling  :start-with 'cases)
(in-grammar 'drilling)

(defclause cases := (repeat+ <case>
           :question "Another case? ")
       :comment  "        WELL  DRILLING  CASE  BASE
ACQUISITION~%")

(defclause case :=
  (seq <contact> <drill-bit> <well-strata> (optional
<remark> :question "Any further remarks on this case?
"))
```

```
  :comment "Acquiring a new case...~%~%A case consists
of the drill bit used, together with the depth of each
~%rock stratum encountered.")

(defclause contact := (seq 'contact <contact-first-
name> <contact-last-name>)
       :comment "Enter the name of the contact person
for any questions relating to this case.")

(defclause contact-first-name := (a-symbolv)
:question "What was the first name of the contact per-
son?")

(defclause contact-last-name := (a-symbolv)
:question "What was the last name of the contact per-
son?")

(defclause well-strata :=
  (repeat+ <stratum> :question "Another stratum? ")
  :comment "Acquiring the rock composition of the dif-
ferent strata, starting from ground level.")

(defclause stratum :=
  (seq 'stratum :rock <rock> :depth <depth>))

(defclause rock := (one-of 'chalk 'clay 'sandstone
'granite)
      :question "What was the rock type?")

(defclause depth := (an-integer-betweenv 1 2000)
:question "What was the depth (in metres) of this rock
        stratum?")

(defclause drill-bit := (seq 'bit-type
(one-of 'bit-type-8765 'bit-type-8760b 'bit-type-8543))
      :comment "Which bit was used for drilling?")

(defclause remark := (a-stringv)
     :comment "Please enter your remarks as a string.")
```

APPENDIX 2

A run of COCKATOO using the above Wells script

```
WELL DRILLING CASE BASE ACQUISITION
Acquiring a new case...
A case consists of the drill bit used, together with the depth of each rock stratum
encountered.
Enter the name of the contact person for any questions relating to this case.
What was the first name of the contact person?
Input a value: Derek
What was the last name of the contact person?
Input a value: Sleeman

Which bit was used for drilling?
The possible values are:
  A:  BIT-TYPE-8765   B:  BIT-TYPE-8760B  C:   BIT-TYPE-8543
Which value? : a

Acquiring the rock composition of the different strata, starting from ground level.
What was the rock type?
The possible values are:
A:  SHALE:   B:  CLAY:  C:  CHALK   D:  GRANITE   E:   OTHER
Which value? : a
What was the depth (in metres) of this rock stratum?
Input a value: 100

Another stratum? y
What was the rock type?
The possible values are:
A:  SHALE:   B:  CLAY:  C:  CHALK   D:  GRANITE   E:   OTHER
Which value? : b
What was the depth (in metres) of this rock stratum?
Input a value: 200

Another stratum? y
What was the rock type?
The possible values are:
A:  SHALE:   B:  CLAY:  C:  CHALK   D:  GRANITE   E:   OTHER
Which value? : c
What was the depth (in metres) of this rock stratum?
Input a value: 300

Another stratum? y
What was the rock type?
The possible values are:
A:  SHALE:   B:  CLAY:  C:  CHALK   D:  GRANITE   E:   OTHER
Which value? : d
What was the depth (in metres) of this rock stratum?
```

```
Input a value: 400

Another stratum? n
Any further remarks on this case?  y
Please enter your remarks as a string:   "Aberdeen Bay - case 1"
Another case? n

 (((CONTACT DEREK SLEEMAN) (BIT-TYPE BIT-TYPE-8765)
  ((STRATUM ROCK CHALK DEPTH 100) (STRATUM ROCK CLAY DEPTH 200)
    (STRATUM ROCK SANDSTONE DEPTH 300) (STRATUM ROCK GRANITE
DEPTH 400))        "Aberdeen Bay - case 1")
```

APPENDIX 3

A run of COCKATOO with an Instrument Description script (slightly revised)

```
An instrument consists of a name and a type.
What is the name of the instrument?: Violin

A chordophone has a :  neck
A aerophone needs : air AND reed(s) AND holes
A membranophone needs:  skin.
Is the instrument played by  strings(chordophone),  air/wind(aerophone)  or
stretched skin(membranophone)?
The possible values are:
  A:   CHORDOPHONE   B:   AEROPHONE   C:  MEMBRANOPHONE
Which value? : a

Does the instrument have a neck?
The possible values are:    A:  HAS-NECK    B:  HAS-NO-NECK
Which value? : a

How many strings does the instrument have ?
The possible values are:  A:  FOUR-STRINGS  B:  SIX  C:  MORE-THAN-SIX
Which value? : a

How big is the instrument?
The possible values are:   A:  BIG  B:  MEDIUM   C:  SMALL
Which value? : c

What is the pitch of the instrument?
```

```
The possible values are:    A:  HIGH    B:  MEDIUM    C:  LOW    D:  ALL-
PITCHES
Which value? : a
Any further remarks on this instrument? n
Another instrument? y
.........................................
.........................................
((VIOLIN (HAS-NECK FOUR-STRINGS SIZE:SMALL PITCH:HIGH))
((OBOE (WOOD NEEDS-REED))
(DRUM (WOOD HIT-WITH:OTHER))   )
```