Requirements For a Knowledge Representation System

(and a personal reflection on Algernon)

Working Note 10

Peter Clark
Dept. Computer Science
Univ. Texas at Austin
Austin, TX 78712, USA

1 Introduction

This document provides a set of general requirements for a knowledge representation system, and explores some of them in more detail. By *knowledge representation system*, we mean the complete package of:

- a **knowledge representation language**, with clearly defined syntax and semantics.
- an **inference engine** which can perform reasoning with a particular representation encoded in that language, to answer questions.
- a knowledge base (KB) **development environment**, possibly including knowledge editing tools, debugging tools, and graphics.

2 Requirements

Several classes of requirements have been identified:

1. **Expressiveness:** The KB language should be expressive enough for the knowledge engineer to say most of what he/she wants to say.

2. **Naturalness:** The representation language should be syntactically friendly, so that a knowledge enterer can both express what he/she wants in a fairly natural way.

3. **Inference:** The system should be able to infer answers to a broad range of questions, minimizing the degree of incompleteness in its inference process.

4. **Semantic Clarity:** The KB expressions should have a clear and well-defined semantics.

5. **Efficiency:** The system should be both time- and memory-efficient, so that it can answer questions in a "reasonable" length of time.

6. **Scalability:** The system's performance should degrade gracefully as the knowledge-base becomes large. It should function with "reasonable" response time with a large knowledge-base loaded in.

7. **Explanation of Inference:** Ideally, a knowledge-based system should not just infer answers to questions, but also be able to explain **how/why** a particular answer was concluded.

8. **Meta-Reasoning ('Introspection'):** The system should be able to introspect on its own knowledge – in other words, not just apply rules but be able to retrieve and manipulate them as objects in their own right. For example, the system should be able to describe, as well as apply, what it knows. This is essential for many tasks, such as:
   - description generation, for example as performed by the Knight system [9]). Knight would generate answers to questions such as "Tell me about cactus plants".
   - natural language processing (NLP). NLP systems often need to know what sort of things can fill different roles, in order to disambiguate sentences (eg. the agent in an eating-event is an animate-object, the patient is a food). Answering such questions requires manipulating, rather than applying, the KB rules, relations, and constraints.
   - building qualitative models, for example as performed by Tripel [12]). Tripel needed to introspect to find qualitative rules, rather than do inference with them, to construct is models.

9. **Support for Knowledge Entry:** Ideally, the KB would not be just a passive database, but would "react" as the knowledge engineer enters data, for example, it might:
   - identify inconsistencies which the knowledge engineer has introduced
   - help the knowledge engineer debug/correct rules if an inconsistency has been introduced (NB. the fault may not always be with the last item entered)
   - warn the knowledge engineer of apparently "unusual" statements which he/she has made
   - constrain what the knowledge engineer can enter (eg. provide templates) so that he/she can avoid entering inconsistencies in the first place.

10. **Contexts and Knowledge Encapsulation:** A system should support the encapsulation of "coherent" collections of rules together into a single conceptual unit. Such packaging is useful for:

    Multiple Modeling: If the system is to allow alternative, possibly conflicting representations to co-exist, some mechanism is needed to encapsulate and isolate those different representations.

    Compositional Modeling: A model fragment includes a rule set expressing relations between variables. Thus some mechanism for grouping rules into sets like this is needed to do compositional modeling.

    Description Generation: A problem in description generation is to identify which information is relevant to include. If some mechanism for packaging is available, the knowledge engineer can make explicit which rules are "about" a particular concept (by packaging them together), thus helping in the description generation task.

    Knowledge Reuse: In a similar vein, it should be possible to package rule sets which axiomatize reusable, conceptual systems, eg. a general model of containers, production, transportation. These are not necessarily all the rules with container (say) as the first argument of their consequent.

11. **Extensibility:** It should be easy to add features or functionality. For example, if there was a need to add in reasoning with 'direct' representations (eg. occupancy grids) it could be included easily.

12. **Foreign System Interface:** It should be able to link the system to external systems, in particular to databases and the World-Wide Web. This requirement may be simply a matter of having an accessible foreign function interface.

13. **Graphics:** A KR system should have a friendly graphical editor to enter, view and apply knowledge.

14. **Robustness:** It should be bug-free.

15. **Portability:** It should be easily portable among a variety of platforms.

16. **Documentation:** It should be well documented.

17. **Cost:** Ideally it should not be expensive to purchase or licence.

3 Expressiveness, Naturalness and Inference

In this Section we explore three of these issues – expressiveness, naturalness and inference – which are fundamental to a KR system.

As a preface, we note that there is often a tension between naturalness and suitability for inference. On one hand, the knowledge engineer's conceptualization of the world should have a simple mapping to the syntactic structures in the KB. On the other, these structures should support the easy design and implementation of an inference engine. These two goals can often conflict, and this tension has been a major challenge in the Botany KB project.
There are two potential sources of unnaturalness in a KR language:

1. The basic syntax of the KR language may be unnatural, for example users may be uncomfortable with Prolog-like for FOL-like expressions. For example, a user may prefer to write

   ```
   (Elephant
     (parts (Trunk)))
   ```

   rather than (say):

   ```
   (forall ?x ((isa ?x elephant) -> (exists ?y (isa ?y trunk) (parts ?x ?y))
   ```

2. A fact which is apparently simple to the user may expand to a large expression in the KR language. For example, the formal expression of "Cars have 4 wheels." or "Pressure is proportional to temperature." ma be a complex logical formula, rather than `(wheels car 4)` or `(proportional-to temperature pressure)`.

In both these cases, it may be possible to hide some of this syntactic unnaturalness: Some 'syntactic sugar' could be used to hide logic-like expressions, and some form of macros could be used to allow logically complex expressions to be stated succinctly.

Of course, naturalness is a double-edged sword: natural expressions are often subtly ambiguous, and so using a more natural syntax may result in more errors and mis-statements to be made, as has sometimes occurred with use of the KM language. A more 'active' knowledge entry tool, which reacts to the knowledge engineer's statements, would help alleviate this problem. Also, a natural syntax does not save the knowledge engineer from learning formal logic; he/she still needs to understand what he/she has said, whatever the language used.

We now examine several specific representational issues, which pose challenges for expressiveness, naturalness and inference in a KR system. Each subsection is prefaced with some example sentences which illustrate the concept(s) which are difficult to encode.

3.1 Defeasibility

In this Section, we consider two aspects of defeasibility:

Representation: What are the semantics of the declarative assertions in the KB? In particular, should a KB expression such as `forall X exists Y f(X)` $\rightarrow$ `g(Y)` translate to an expression in first-order logic or a default logic (eg. [11])?

Inference: How can we perform non-monotonic inference with those assertions?

3.1.1 Defeasible Representations: "To Augment or Over-ride?"

1. "Typically birds fly, but penguins don't."
2. "People's pets are any animal except lions."

Should a 'more specific' rule (ie. a rule associated with a more specific set) over-ride or augment a 'more general' rule (ie. a rule associated with a superset)? In many cases, over-riding is highly desirable – it allows complex concepts to be concisely described as a set of overly-general assertions plus exceptions. However, in other cases, it is not desirable. Consider first the canonical Tweety-the-penguin example:

```
Bird
----
  flies? True

Penguin
-------
  flies? False
```

In this case, we would like the `flies?` rule for penguins to over-ride the rule for birds. Now consider a second example:

```
Person
------
  drives: Car
          ---
            type: Domestic

Man
---
  drives: Car
          ---
            speed: Fast
```

In this case, we would like the information about a man's car to *augment* information about a person's car, not over-ride it. We'd somehow like to merge the car information from `Person` and `Man` to form a composite description of mens' cars.

KM and Algernon both adopt an "augment" policy (hence will answer {`true,false`} to the question "Does Tweety-the-penguin fly?"). KQL adopts an "over-ride" policy (hence will fail to answer the question "What type of car does a man drive?"). It seems that no single policy is 'best' – rather, the user should be able to select one or the other on a case-by-case basis.

Although assertions in a KQL KB are defeasible (ie. are over-ridden by more specific assertions), its inference engine is unsound as it uses negation-as-failure and is monotonic – the KQL interpreter is unable to retract assertions which may become invalid as more information becomes avaiable. For example, if Tweety isn't known to be a penguin then the KQL interpreter will conclude (if asked) that Tweety flies. If later it is asserted that Tweety is a penguin after all, it will not undo the flies assertion[1]. This mixture of defeasible assertions with monotonic inference may seem like trouble (eg. see [3]), but in practice trouble has rarely arisen.

3.1.2 Defeasible Reasoning: Drawing Tentative Conclusions

3. "Typically, liquids are applied with a brush."
4. "Typically, leaves are green."

KM, KQL and Algernon are all monotonic systems – they only compute the consequences logically implied [2] by the information that they know, and cannot retract assertions.

In many cases it would be highly useful to draw tentative conclusions, perhaps using some probability measure to express preference. A particular situation where this commonly arises is with "automatic classification", whereby classification rules allow a general representation to be specialized as more information becomes available. Consider the following rules:

1. to apply some object P to some object O, an instrument I is needed.
2. if P is a liquid, then I is a brush.
3. if P is a nail, then I is a hammer.

Rule 1 is a (very simple) general model of the concept "apply". Rules 2 and 3 are classification rules allowing this general model to be specialized as more information becomes available. For example, if the system is considering an instance of an apply where P is paint, then it can conclude that I must be a brush (as paint is a liquid). While this sounds fine, the lurking problem is that these classification rules are overly strong; in this example, other things may be used to apply paint (say) also. Rather, we would like to express:

2a. if P is a liquid, then I is probably a brush...
2b. ...but it could also be a roller or a spray.

Making and reasoning with such plausible assertions is a thorny issue; however it would be highly useful in a KR system. It may be possible to admit such plausible reasoning in a constrained fashion, rather than doing full non-monotonic inference.

3.1.3 Probabilistic Reasoning

Another approach to defeasibility and uncertainty would be to use some probabilistic representation, eg. Bayes' nets. Such reasoning mechanisms could probably be encoded within the deductive framework of a logic-based KR system. Such mechanisms might be useful if a lot of data is available for computing the required probabilities.

3.2 Representing Constraints

5. "There are at least 2 destinations for distribution, and typically several."
6. "Fred is at least 21 years old."

[1] Operationally, the "Tweety flies" conclusion is cached on the Tweety instance, and hence is considered to over-ride the more general penguin rule.

[2] or an approximation of this, if negation-as-failure is used

7. "Silty soil consists of <12% clay, 80%-100% silt, and <20% sand."
8. "A rectangle's area = height × width."
9. "The total of the % of clay, silt and sand in SiltySoil is 100%."
10. "A square's height = its width."
11. "One sperm fertilizes the egg and the other sperm fertilizes the endosperm." (Issue: enforce non-coreferentiality)

Much common-sense knowledge is in the form of constraints. As illustrated by these examples, there are many different types of constraint knowledge. Ideally, a KR system should be able to monitor and enforce these.

The nearest example of a system which does this is the programming language LIFE [1]. LIFE maintains a run-time database of active constraints (called *residuations*), which are repeatedly checked to ensure that none has been violated, and to compute values if sufficient information becomes available (eg. `height` can be computed from the constraint `area=height×width` given `area` and `width`). This is implemented efficiently by tagging variables with the constraints that they participate in, and hence LIFE can quickly identify which constraints to check as variables' values become known. Despite this clever mechanism, however, constraint checking is intrinsically a computationally expensive process.

For simple invariants such as "width = height", it is possible to enforce the constraint using forward-chaining rules. For example in Algernon we could write:

```
(:rules Squares ((height ?s ?h) -> (width ?s ?h))
                ((width  ?s ?h) -> (height ?s ?h)))
```

The approach becomes more difficult though with complex constraints, for example a range constraint ("Fred is at least 21 years old."), or one where a variable cannot be easily isolated (eg. isolating x in $y = x^2 + x + 1$). In these cases, as in LIFE, a 'constraint database' would be needed to ensure constraints were not later violated.

The other examples at the start of this section illustrate other types of constraints which might be expressed:

- Constraints (rather than definite assertions) about a relation's cardinality
- Numeric constraints
- Constraints which span multiple relations about an object (eg. height and width)
- Enforcing non-coreferentiality. Again in LIFE a constraint such as `?x` $\neq$ `?y` will residuate; if at any point an attempt is made to set `?x` to equal `?y`, the constraint will be checked and hence the computation will fail (thus treating negation as FOL negation, rather than negation-as-failure).

3.3 Coreferentiality and Unification

3.3.1 Enforcing Coreferentiality

12a. "People drive exactly one car."
12b. "People drive domestic cars."
12c. "Men drive fast cars."
12d. "Fast, domestic cars are expensive."
12e. "Is Fred-the-man's car expensive?"

Sometimes, several rules may contribute information to the same instance. Consider the earlier example about people and men driving cars; we could encode this in Algernon (say) as:

```
(:slot  drives (People Vehicles) :cardinality 1)
(:rules People ((drives ?p ?c) <- (:a ?c (isa ?c Cars) (type ?c Domestic))))
(:rules Men    ((drives ?p ?c) <- (:a ?c (isa ?c Cars) (speed ?c Fast))))
```

If we ask "What does Fred drive?", then the system would generate two instances of Cars[3], but not merge this information together (even though it 'knows' the instances are coreferential, from the `:cardinality 1` declaration). Thus some inferences may be missed: for example if a rule stated that fast, domestic cars are expensive, the inference engine would be unable to conclude that Fred's car must therefore be expensive.

In conceptual graphs, merging differing descriptions of the same instance is a fundamental operation – called a "maximal join". This is also the basic operation in the programming language LIFE [1], and is called "psi-term unification".

It's possible that an equality predicate could be defined in Algernon which performed a similar operation, named (say) `(equals ?x ?y)` or `(:unify ?x ?y)`. This predicate would be an assertion rather than a query, and would prompt Algernon to unify two individuals together using a standard, recursive unification algorithm.

3.3.2 Coreferentiality and Sets

13. "A person has a head and two legs."
14. "There are exactly three (types of) raw materials for photosynthesis."

Assuming non-coreferentiality of instances can often cause confusing behaviour with multi-valued relations, eg. `parts`, in particular when the user would like to ask questions of the form "how many <X> are there?". Consider encoding the above statement as:

```
(:rules People
        ((part ?_ ?h) <- (:a ?h (isa ?h heads) (name ?h "head")))
        ((part ?_ ?l) <- (:a ?l (isa ?l legs) (name ?l "leg")))
        ((part ?_ ?l) <- (:a ?l (isa ?l legs) (name ?l "leg"))))
```

The third rule is considered redundant in both KM and Algernon (it apparently duplicates the second rule), and hence asking for the parts of a person will return

```
parts = head1, leg1                     ; a one-legged person?
```

If the query is reissued, then the rules fire a second time to give:

```
parts = head1, leg1, head2, leg2        ; a two-headed person?
```

This behaviour is technically correct – saying the person's got one leg doesn't preclude that he/she has another (unreported) leg, and giving parts = {`head1,head2`} is fine if the two heads are coreferential. But of course it would be better if the system recognized the coreferentiality, or the user could declare the coreferentiality, so that these slightly obscure answers aren't produced.

3.4 Representing Disjuncts

15. "Turbulent flow tranports either a liquid or a gas."

Disjuncts are easy to state but often difficult to handle in inference. A simple 'trick' is to reify the disjunctive class as a concept in it's own right (eg. `Liquids-or-gases` with specializations `Liquids` and `Gases`), but of course this does not completely solve the issue. Rather, some form of non-monotonic reasoning is again needed to tentatively assume on or the other and then compute the consequences.

[3]Or consider this a knowledge entry error, in Algernon 2.

3.5 Second-order Expressions

16. "Joe believes that Pete has a car."
17. "If you pick up a block, then you'll hold it as a result."

For some reasoning tasks, such as planning, it is necessary to treat KB assertions as objects in their own right. This requires being able to use expressions as the arguments in relations, for example

```
(believes Joe (has Pete Bruces_car))
```

There are two schools of thought on how knowledge like this should be represented. The first (the modal logic camp) suggests that some modal logic is required (eg. [10]), the second (the syntactic theory camp) suggests that this form of reasoning can be handled within first-order logic, where syntactic expresssions are simply another object in the world (eg. see [7] for an inspiring article). I personally find the syntactic approach most appealing. An example of this, where the expression is reified, would be to create the class `Expressions` wiith applicable relations `arg1`, `arg2` and `functor` (say):

```
(:a ?e (isa ?e Expressions)
       (functor ?e has)
       (arg1 ?e Pete)
       (arg2 ?e Bruces_car))
(believes Joe ?e))
```

Syntactic sugar could be used to make the syntax more pallatable.

4 A Personal Reflection on Algernon

Finally, I present here some personal reflections on Algernon. As a preface, I've been struck by the strong similarity of the underlying approach to inference used in the four KR systems I'm familiar with (KM/KQL [4], CycL [8], KRL [2], and Algernon [6]). In various syntaxes, these systems all operate using the general mechanism of generating Skolem constants to denote individuals, and then applying rules to those individuals (possibly creating new individuals in the process) to infer answers to questions.
I now present some more detailed comments on Algernon with respect to the requirements raised in Section 2.

1. **Expressiveness:** Algernon's language is based on first-order logic, and in general seems sufficient for expressing much common-sense knowledge. For many representational problems, the issue concerns *how* to best conceptualize the problem in firt order logic, rather than requiring a more expressive language.

   To me, the most problematic issue relating to expressiveness is Algernon's inability to accomodate knowledge as "overly-general rules + exceptions" (Section 3.1.1). Although the use of prototypes can accomodate simple cases (where one slot-value can over-ride another), this solution would be difficult to generalize to handling more complex rules and relationships.

   A second requirement is the ability to represent second-order expressions (Section 3.5). Following Haas [7], I believe these representational problems can be handled within first-order logic, and hence within Algernon's framework, although some syntactic sugar would be nice.

2. **Naturalness:** Algernon's syntax is probably rather unnatural for many people; Algernon could perhaps be viewed as a "knowledge programming language". Although I'm comfortable with its syntax, I'm uncomfortable with the way conceptually "simple" knowledge expands into a large set of clauses (as illustrated in [5]). There of course may be other ways in which CGs can be expressed in Algernon; some more thought on this issue is needed. In general, though, a higher-level "knowledge language" which compiled down into Algernon rules would be invaluable, although this would complicate debugging and maintaining a KB. This would require identifying common, "simple" statements at the knowledge level (eg. "cars have four wheels") and how they should translate to Algernon rules.

3. **Inference:** Algernon's inference is clearly defined and really works. This I think is a great strength of the system, to have a fully functional inference engine at hand. Its range of special forms (eg. `:retrieve`, `:bind`) seems to handle most representational requirements. It overcomes some of the incompleteness problems in KM (eg. see the demo file `cg3.lisp`).

   The main complication which arose with Algernon's inferencing is its handling of coreferentiality (Section 3.3); often, during debugging the demos, multiple solutions were returned when only one was intended, and prolific use of `:forc` and `:cardinality 1` was needed to supress this. Again, there may be better ways of overcoming this.

4. **Semantic Clarity:** Another good feature of Algernon is its semantic clarity – it has clear and well-defined semantics, and was easy to work with, and compares favorably with the sometimes unclear or situation-dependent semantics in KM.

5. **Efficiency and Scalability:**

   Algernon has two nice properties for efficiency: First, associating rules with sets and second, using access paths. These two make Algernon perform well-focussed inference, and avoids the impracticalities which would arise if the KB was expressed in Prolog (say). KM/KQL has the equivalent of these two features also, and, from my limited reading, it appears KRL may also have similar features.

   There are two potential problematic areas for efficiency/scalability, which I haven't fully explored:
   - its policy of following many alternatives in parallel may be potentially expensive
   - conceptual graphs expand into a large number of rules [5]. While this improves Algernon's completeness, it is a potential problem for scalability.

   Continuations in Algernon are another potential scalability problem, but in fact this is not a strong concern to me as my natural inclination is to populate a KB with backward-chaining rather than forward-chaining rules.

6. **Remaining Items** The remaining items listed in Section 2 are all desirable properties that Algernon either has or could be added to Algernon. For a KR system to be used in a larger (multi-user) environment, a Web-based interface for knowledge entry, debugging and querying would be a very nice thing to have.

References

[1] H. Ait-Kaci and A. Podelski. Towards a meaning of LIFE. *Logic Programming*, 16:195–234, 1993. (also available as http://www.isg.sfu.ca/ftp/pub/hak/prl/PRL-RR-11.ps.Z).

[2] D. Bobrow and T. Winograd. An overview of krl, a knowledge representation language. In R. Brachman and H. Levesque, editors, *Readings in Knowledge Representation*, pages 264–285. Kaufmann, CA, 1985. (originally in Cognitive Science 1 (1), 1977, 3–46).

[3] I. Brachman. 'i lied about the trees', or, defaults and definitions in knowledge representation. *AI Magazine*, 6(3), 1985.

[4] P. Clark. KM/KQL: Syntax and semantics. (Internal document, AI Lab, Univ Texas at Austin. http://www.cs.utexas.edu/users/mfkb/manuals/kql.ps), 1996.

[5] P. Clark and B. Kuipers. Conceptual graphs in algernon and km. (Internal document, AI Lab, Univ Texas at Austin), 1996.

[6] J. M. Crawford and B. J. Kuipers. Algernon – a tractable system for knowledge-representation. *SIGART Bulletin*, 2(3):35–44, June 1991.

[7] A. R. Haas. A syntactic theory of belief and action. *Artificial Intelligence*, 28:245–292, 1986.

[8] D. Lenat and R. Guha. *Building Large Knowledge-Based Systems.* Addison-Wesley, MA, 1989. (Also see http://www.cyc.com/public.html).

[9] J. C. Lester and B. W. Porter. Developing and empirically evaluating robust explanation generators: The knight experiments. *Computational Linguistics*, 1996. (to appear). http://www.cs.utexas.edu/users/mfkb/papers/coling.ps.Z.

[10] R. C. Moore. A formal theory of knowledge and action. In J. R. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex, NY, 1985.

[11] R. Reiter. Nonmonotonic reasoning. In H. E. Shrobe, editor, *Exploring Artificial Intelligence*, pages 439–481. Kaufmann, CA, 1988.

[12] J. W. Rickel. *Automated Modeling of Complex Systems to Answer Prediction Questions.* PhD thesis, Dept CS, Univ. Texas at Austin, 1995. (Also available as Tech Rept AI-95-234). http://www.cs.utexas.edu/users/mfkb/papers/rickel-phd.ps.Z.