A LOGIC-BASED ARCHITECTURE FOR KNOWLEDGE MANAGEMENT

Damian Black & John Manley

Bristol Research Centre
Hewlett-Packard Laboratories
Filton Road, Stoke Qifford
Bristol BS12 6QZ, England

Abstract

We define and discuss an architecture for a knowledge base management system (KBMS) to support the efficient manipulation of large and diverse bodies of knowledge. The architecture has been formulated within an amalgamated language-metalanguage framework. All aspects of the system possess a logical semantics. Our emphasis is on modularity, the organization of theories and their metatheories within the system theory, and the clean integration of multiple representation languages. The expressivity of the architecture is discussed and an implementation based on an extended Prolog reported. Its utility is currently being investigated in a number of diverse applications.

1. Introduction

The work reported here began two years ago with the aim of addressing the problems of complexity and efficiency that arise in the management of large and diverse bodies of knowledge. Logic programming is our starting point. Although Prolog is the most mature logic programming language, it has a number of shortcomings that make it unsuitable in its current form as the basis for a KBMS. We have attempted to overcome these failings.

What are the attributes of a KBMS that we believe are important? It must:

- o possess simple semantics
- o support and integrate multiple representations
- o encourage reusability of knowledge
- o support a logical model of updating
- o efficiently handle very large amounts of data
- o provide a general basis for a range of flexible and powerful tools
- o subsume RDBMS capabilities

These attributes are further discussed in [Sharpe ef a/ 86]. The key to overcoming the problems with Prolog is using the Horn clause language as its own metalanguage - the strict separation of logic from metalogic expressed in an amalgamated language and metalanguage [Bowen & Kowalski 82]. However, this amalgamation merely provides a framework within which we must define a KBMS architecture that meets our stated objectives.

Our architecture incorporates theories as first-class data-objects which provide the basis for the representation, storage, retrieval and updating of knowledge. The KBMS provides an organization of theories that controls their access and interactions.

2. The KBMS Architecture

In our view a knowledge base comprises not only the data but also the metadata - in other words knowledge about how to handle that data. All this knowledge is partitioned into theories and organized within a single system. Theories are referred to via terms that act as names for those theories. There are two basic interfaces to named theories: proving and updating. The former is used for handling all queries, and the latter for handling all updates.

The representation of knowledge within a named theory is regarded as hidden. The theory is effectively a black box with only its metatheories (interpreters and assimilators) having direct access to the contents of the theory. As a consequence of this structuring, multiple representations of knowledge within a single knowledge base is possible. Integration of the separate theories is achieved by allowing variables of a query in a named theory to be bound by the interpreter of that theory to terms within the theory. We therefore require a global convention across the whole knowledge base for the meaning of such terms.

2.1 The System

In our architecture, the entire knowledge base is represented by a distinguished theory known as the system. This theory defines the relationship between names of theories, the theories themselves and the names of their metatheories (interpreters, assimilators and attributes). Whenever a query is made in a named theory that theory must be retrieved from the system - performed by querying the system. The relationships between names and theories is expressed using the full power of Horn clauses, thereby enabling the representation of complex many-to-many relationships.

Since the system is itself a theory, it can easily be extended by including extra axioms to define, say, a variety of methods for retrieving named theories and choosing any metatheories required. Another effect of this architectural decision is that updates to the system are handled like updates to any other theory. Consequently, a new system results from any update. Potentially access to all previous versions of the system is possible.

2.2 Theories

Theories are built from bags of Horn clauses. Unit clauses can readily support a variety of representation languages. Of course, if queries are to be handled for a given representation then an interpreter must be defined. Every clause in a theory has a unique identifier for possible use in metatheories. AM theories in the system have associated with them three sets of metatheories: interpreters, assimilators, and attributes. Each of these sets contains one or more members but if there is more than one member then inference within the system is needed to choose between them. All metatheories, being theories, similarly possess three associated sets. However, to avoid the apparent infinite regress, recursion in each of the three sets quickly terminates with built-in metatheones. For instance, efficiency considerations mean that more than two levels of interpretation are rarely used in practice. The built-in interpreter is Prolog.

2.3 Interpretation

All theories are queried via a standard predicate interface. This interface takes a theory name and a set of goals: kbms prove(System,TheoryName,Goals). The system is consulted to find the current version of the named theory and the name of its interpreter. If there is more than one interpreter for the named theory then inference within the system is used to choose between them. The KBMS then attempts to prove that the interpreter can solve the goals. This causes a recursion - due to the need to find an interpreter for the interpreter - terminated by the built-in interpreter.

A theory's interpreter may be one of the standard theorem-provers, but, as is well-known, uniform proof procedures are very inefficient [Bundy et al 79]. It is desirable to make use of domain-specific control in order to direct the proof especially when large bodies of knowledge are involved. By allowing each theory to have its own interpreters we are able to carry out theory-specific inference control. Inference control can make use of both syntactic and semantic constraints, for example type information and the number of sub-goals in a candidate clause [Gallaire & Lasserre 82, Bundy et al 79]. The primary use of attribute theories is to allow particular theory-dependent information to be abstracted out of interpreters and assimilators thereby making them theory-independent and hence reusable. Non-standard interpreters have to be user-supplied; they have to be accessible to the querying interface and "understand" (supply a semantics for) the representation of the theories for which they act as theorem-provers. Interpreters have the full power of the Horn clause metalanguage at their disposal and they operate on theories via a number of lower-level, built-in KBMS primitives. Theory-to-theory linkages are obtained by including goals in one theory that require a proof in another named theory.

2.4 Assimilation

All assimilation proceeds via a standard interface: kbms update(System .Theory Name.Updates,New System). In a similar fashion to interpretation, all assimilators must understand, and translate into, the representation of the theory they are updating. Updates are specified as a list of actions - all actions are eventually reduced into primitive, declarative add and delete operations on the theory. The simplest form of assimilator is one that merely carries out all additions and deletions without any kind of check. Alternatively the assimilator can enforce integrity constraints. There are two basic approaches to updating under constraints: to carry out a list of actions, checking on the consistency of the final theory; to carry out the whole set of relevant checks after the addition, or deletion, of each clause. The former method has the advantage of ignoring unavoidably inconsistent transitional states of the theory. If an illegal update is attempted, the assimilation fails and all related updates will be undone during backtracking. The checks advocated by Kowalski [Kowalski 79] can be incorporated into an assimilator, but they prove to be very expensive. A second set of checks might be called domain-dependent, syntactic and semantic integrity constraints. Under this heading one might place constraints such as: predicate P must always be binary; the second argument of predicate P must always be less than 100. As with interpreters this type of theory-dependent information can be abstracted out into the theory's attribute theory thereby making the assimilator applicable to more than one theory.

The update model used is completely declarative. This means that any update (addition or deletion) to a theory results in a new theory. Both the previous version and the new version are accessible as long as references to both still exist. This can be achieved with very little time and space overhead. Since the system in some sense contains all theories, an update to a theory is also an update to the system. As the system is also a theory, a new system is created whenever any update occurs. Again, all previous versions of the system are available as long as they are still referenced.

The system also has an assimilator. A typical task might be to check that when adding a new theory to the system all of its nominated metatheories also belong to that system.

3. An Implementation

We have implemented a KBMS, KBMSO, adhering to the principles stated earlier. It possesses all the functionality described above and was built from an extended Prolog developed to meet our requirements. The extensions provide the theory as a first-class data-object supporting declarative operations. All of these operations are backtrackable leaving the state of the KBMS unchanged. In implementing declarative updating, our design decision was to maximize the speed of accessing versions of a theory at the expense of space. However updates to the most recent version of a theory are handled incrementally with the minimum of space overhead as a single theory datastructure contains all such incrementally produced versions. When an update is made to a version which is no longer the current one copying occurs.

The execution speed of querying Horn clauses stored in theories, using Prolog as their interpreter, runs at about two-thirds the speed of the native Prolog system. There is no space overhead, and little time overhead, incurred in querying previous versions of a theory.

4. Expressivity and Limitations

There are several techniques that have been reported as being important in the management of large knowledge bases (Bowen 85, Kauffman & Qrumbach 86, Furukawa et al 84]. We believe that these are really secondary to the fundamental problem of organizing information into a rigorous logic-metalogic framework. With the right architecture, they all become artefacts of particular representation methodologies or specialized logic programming devices.

4.1 Inheritance

Inheritance has been put forward as a primary structuring concept for the design of modularized knowledge bases. It has also been used to tackle problems of representation (such as in default reasoning). It is undoubtedly useful in certain cases, but it is an imprecise term with many interpretations. In introducing any form of inheritance into our architecture we must not violate our basic design philosophy of named theories behaving as private queriable bodies of knowledge. This precludes any assumptions being made about the representation of such theories. Direct inheritance of clauses of one theory by another is therefore not allowed since it would violate this principle for two reasons:

1) it would create a reliance on understanding the representation of clauses in other inherited theories;

2) a theory's assimilator would no longer have absolute control over the contents of that theory and would be unable to enforce any associated integrity constraints.

Even when two theories have a common representation, semantic inconsistencies can still arise between the theories. The situation is even worse when non-unit clauses are used. Trying to overcome these problems by defining schemes for selectively overriding predicates is unsatisfactory: such techniques do not scale up for use in large, diverse knowledge bases. In spite of this other metalanguage-based architectures have placed a strong emphasis on provision of inheritance facilities [Kauffmann & Grumbach 86, Furukawa *et ai* 84]. A factoring of knowledge can be straightforwardly achieved by modularization into named theories, and the use of the *kbms prove* predicate to achieve explicitly what inheritance achieves implicitly, at the same time adhering to the ideal of black-box theories.

The system allows the definition of arbitrary relationships between names and theories, bounded only by the expressivity of Horn clauses. This leads to a flexible method where an inheritance lattice can be readily described. One form of inheritance is merely theory selection where a goal is proved in one member of a set of theories that are related through the inheritance lattice. Backtracking on failing to satisfy the goal in one candidate theory causes attempted satisfaction in the next.

Another form of inheritance is naturally incorporated into a theory's interpreter. Its principal disadvantage is that it admits the possibility of using semantically inconsistent knowledge. When solving a goal, the interpreter tries first to satisfy that goal in the named theory. If this fails, then instead of backtracking to the previous goal, it queries the system to find the name of a theory from which this theory inherits. It then attempts a proof of the current goal in this alternative theory, which of course proceeds via its own interpreter. This process can continue until the goal is satisfied or until the goal has failed in all the pertinent theories in the inheritance lattice - backtracking takes care of multiple inheritance. It differs from the proceeding method in that it allows each individual subgoal to be tried in each theory, instead of just the overall goal. Implementing it requires only a minor enhancement to a theory's interpreter. Note that such a scheme does not violate any of the principles stated above.

4.2 Set Operations on Theories

Inheritance can be incorporated into a KBMS with regard to the principle of theories being treated as black boxes. The same is not so obviously true of set operations on theories. The union of two theories should not simply be constructed by forming a new theory that is the sum of the two sets of clauses - we might be merging two different representations, or even two semantically inconsistent bodies of knowledge. MetaProlog [Bowen & Weinberg 85), however, encourages this formation of theory unions through the provision of a special operator. If we regarded the union operation as being the addition of one theory to another, then clauses of the second could be added to the first via the firsts assimilator. The obvious faults of this method are that the assimilator might not understand the representation of the clauses it is adding, also that the merged theories might still be semantically inconsistent according to constraints in the second theory's assimilator. The same considerations apply to other set operations such as *difference*. Thus it appears that there are no general solutions to this problem.

4.3 Proof Trees

In some cases efficient control of inference requires being able to reason about global aspects of the proof so far - this requires access to a proof tree. However, like set operations, returning proof trees as a result of querying a named theory violates the principle of regarding that theory as a black box. This is because the nature of the proof tree depends heavily on the representation of the theory and the nature of its interpreter. Ideally control information should never be passed in or out of queries. It is the sole responsibility of the interpreter to employ whatever techniques it deems necessary to handle a query. If the formation and use of a proof tree is internal to a particular interpreter, no problems can arise. This is contrary to an approach described in metaProlog [Bowen & Weinberg 85].

4.4 Partial Evaluation

Partial evaluation is a technique whose generality and applicability has been overstated. When there of static information available which affects control of inference, then partial evaluation can be effectively used to remove redundant run-time computation. This gives an elegant model of the process performed by compilation [Kahn 84, Kursawe 86]. However the power of an interpreter lies in controlling combinatorial explosion. This normally requires dynamic information and hence cannot be evaluated in advance. An example is goal ordering based on instantiation levels. Another problem with partial evaluation is that it is time and space consuming. It can be useful in certain restricted situations such as in handling a frequently used class of queries on a relatively static theory. The obvious place in the architecture for a partial evaluator is in the role of an assimilator.

It is possible to incorporate all these aspects into the architecture described. In some cases strongly held ideals will be violated but this is a matter of user responsibility. What is required is a methodology for their use - this is beginning to come from the extensive and varied uses to which the system is currently being put.

5. Summary

We have described the motivation and principles underlying the design of a KBMS architecture. All aspects of the system possess a clear logical semantics. The use of an amalgamated language and metalanguage gives rise to considerable expressivity and flexibility in the representation and manipulation of knowledge. A central feature is the modularization of large bodies of knowledge into theories that can be regarded as black boxes. The use of standard query and update interfaces allows the representation of the knowledge to remain hidden. We have implemented a system (KBMSO) on top of an extended Prolog, and its utility is currently being investigated in domains such as: protein structure determination, co-operative problem solving, commercial database applications and declarative graphics.

References

Bowen, K.A., (1985], "Meta-Level Programming and Knowledge Representation," *New Generation Computing,* 3, 359-383.

Bowen, K.A. and Kowalski, R.A., [1982], "Amalgamating Language and Metalanguage in Logic Programming," *Logic Programming,* Clark, K.L. and Tamlund, S.-A. (eds), 153-172, Academic Press, New York.

Bowen, K.A. and Weinberg, T., [1985], "A Meta-level Extension of Prolog," *1985 Symposium on Logic Programming,* Boston. 48-53, IEEE.

Bundy, A., Byrd, L., Luger, Q., Mellish, C, Milne, R. and Palmer, M., [1979], "Solving Mechanics Problems using Meta-Level Inference," Proc IJCAI-79, Buchanan, B.Q. (ed), 1017-1027.

Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M. and Ueda, K., [1984], "Mandala: A Logic-based Knowledge Programming System," Proc Int Conf Fifth Generation Computer Systems, 613-622.

Gallaire, H. and Lasserre, C, [1982], "Metalevel Control for Logic Programs," Logic Programming, Clark, K.L. and Tarnlund, S.-A. (eds), 173-185, Academic Press, New York.

Kahn, K., [1984], "The Compilation of Prolog Programs without the Use of a Prolog Compiler," Proc Int Conf Fifth Generation Computer Systems, 348-354.

Kauffmann, H. and Grumbach, A., [1986], "Representing and Manipulating Knowledge within 'Worlds'," Proc First Int Conf on Expert Database Systems, Kerschberg, L. (ed), Charleston, 61-73.

Kowalski, R., [1979], Logic for Problem Solving, North-Holland, New York.

Kursawe, P., [1986], "How to Invent a Prolog Machine," Proc Third Int Conf on Logic Programming, Shapiro, E. (ed), London, 134-148, Springer-Verlag.

Sharpe, W.P., Hull, R., Black, D.S., Manley, J.C. and Zaba, S.J. [1986], A Methodology and Architecture for Knowledge Base Management Systems, HPLabs, Bristol Technical Report No. HPL-BRC-TR-86-031.