N90-27302

A KNOWLEDGE BASE ARCHITECTURE FOR DISTRIBUTED KNOWLEDGE AGENTS

Joel Riedesel
MS: S-0550
Martin Marietta Astronautics
P.O. Box 179
Denver, Co. 80201
jriedesel@den.mmc.com

Bryan Walls
NASA, Marshall Space Flight Center
Bldg. 4487 EB12
Huntsville, AL 35812
bwalls@nasamail.arc.nasa.gov

ABSTRACT

In this paper a tuple space based object oriented model for knowledge base representation and interpretation is presented. An architecture for managing distributed knowledge agents is then implemented within the model.

The general model is based upon a database implementation of a tuple space. Objects are then defined as an additional layer upon the data base. The tuple space may or may not be distributed depending upon the database implementation. A language for representing knowledge and inference strategy is defined whose implementation takes advantage of the tuple space. The general model may then be instantiated in many different forms, each of which may be a distinct knowledge agent. Knowledge agents may communicate using tuple space mechanisms as in the LINDA model as well as using more well known message passing mechanisms.

An implementation of the model is presented describing strategies used to keep inference tractable without giving up expressivity. An example applied to a power management and distribution network for Space Station Freedom is given.

1. Introduction

In this paper a tuple space based object oriented model for knowledge base representation and interpretation is presented. The model provides a general knowledge language that is at once expressive and extendable. This allows it to be applied to many different domains including knowledge base management systems for expert system shells and architectures for distributed knowledge agents.

The field of Distributed Artificial Intelligence (DAI) is very complex. Besides the problems involving representing any particular agent, there is a whole new set of problems that are concerned with how multiple agents communicate with one another. This problem is more than just defining a mechanism but also involves protocols. How does one model of communication enhance the ease of solving one problem over another model?

The model presented in this paper supports DAI at a low level. This model presents a framework for defining multiple knowledge agents that must coordinate and cooperate with one another to solve some problem. This is different than most papers about DAI in that most papers are concerned with problems of communication and cooperation protocols. This model defines an architecture that supports the definition and implementation of diverse knowledge agents and their necessary protocols as the problem requires.

1.1. Three Requirements for Representing Distributed Knowledge Agents

There are a number of requirements a model will need to represent distributed knowledge agents adequately. Four interrelated requirements are identified and discussed here: Domain independence and expressivity, control knowledge, and communication.

1.1.1. Domain Independence and Expressivity

Domain independence and expressivity of a model are very closely related. Domain independence is concerned with the ability of the model to represent problems from any domain. The particular representation and storage of knowledge is important to domain independence. However, more than simply being concerned with expressing a problem in the model, domain independence is concerned with the ability of the model to integrate with the various problem domain environments. Expressiveness is concerned particularly with the ability and ease of stating a problem in the model language and not how the problem might have to deal with the environment of the problem.

The requirement for domain independence is concerned with the ability to represent a problem in the model and integrate that problem solution into the problem environment. This really implies that the language of the model must be extensible. It must be possible to enlarge the language using the base language as a start. This includes the ability to change inference mechanisms and define new models of inference. While most languages are concerned with representing data, this language is also concerned with representing control knowledge.

To support this requirement the language of the model presented here is a rule language built using object-oriented programming and extendable using the objects of the language. Rules are a primitive object and are evaluated and interpreted based upon a generic view of data stored in an independent database. A basic rule group object is provided with a forward chaining inference mechanism. A rule group may also be used to control the execution of another rule group thus enabling the definition of new inference mechanisms using the language. Additionally, more specific rule groups may be defined as sub-classes of the base rule group to support different inference mechanisms defined with meta-rule groups.

The other hand of domain independence is expressivity. There are at least two aspects to expressiveness: domain knowledge representation and control knowledge representation.

The problem of expressivity is that as more expressivity is allowed, along with more domain independence, the more intractable a language may become. The basic inference model for knowledge based systems consists of a match-select-fire cycle. The match phase determines those rules which are enabled and may be fired. The select phase selects one rule from the matched rules and the fire phase fires, or interprets, the selected rule. This basic paradigm captures the model of inference that knowledge base systems perform. To make this efficient, there are a number of options to the knowledge base designer. The language may be restricted, for example providing only universal quantification. Various compilation mechanisms may also be incorporated to make the match phase as static as possible. Both of these mechanisms are performed in OPS5 using the RETE network ([7, 8]).

The model presented here provides a large amount of expressivity while moving from the basic inference model presented above to something more tractable such as OPS5 and the RETE network.

To provide a large amount of expressiveness, the language of the model provides a frame system for representing structured knowledge as well as simple facts. Rules may use data from frame knowledge and fact knowledge. In [11], Hayes-Roth presents a number of knowledge categories that are needed for benchmarking different knowledge base systems. The categories fact, rule, class, entity, relation, and structure are provided by the model presented here. The remaining categories are not provided for but are planned as future work.

1.1.2. Control Knowledge Representation

Another requirement is the representation of control knowledge. The interpretation of the knowledge of different knowledge agents will need to be based upon the needs of the different knowledge agents. Some may require forward chaining while some may require backward chaining. Some may require more exotic strategies such as forward chaining with beam search.

There are two aspects of control knowledge identified here. One is the inference strategy used over a rule group and may be provided by defining meta-rule groups or LISP code. The other way of controlling rules is by providing a level of determinism into the ordering of rules themselves ([1, 9]). The way this is done in most expert system shells is by adding variable references into rules that provide control (e.g. IF `[step = 1]` ... `THEN` ...). This makes maintenance of the rules very difficult. An alternative is to use a transition table, allowing any level of non-determinism. Each entry in the transition table is used to index into the next possible rules. This is equivalent to defining a regular expression over a rule group ([9]).

The mechanism provided here is a transition table. As a rule is fired, that rule (its name) is used to index into the transition table to determine what rules are allowed to be fired on the next cycle. If the particular inference strategy being used allows more than one rule to be fired in a particular cycle, each of the fired rules is used to index into the transition table and the resulting lookups are unioned together. Complete non-determinism may be provided by not using a transition table and complete determinism may be provided by specifying only one rule as the next rule for any particular rule fire. Any mix of determinism and non-determinism may be specified using this mechanism. This is discussed further in section 2.

1.1.3. Robust Communication

The last requirement is for mechanisms for communication and coordination between knowledge agents. There is definitely not a consensus on the best mechanism for communication in the literature. The basic mechanism for communication and coordination provided here is a tuple space model based on LINDA ([5, 14]).

The tuple space model allows the addition of knowledge agents without having to modify existing knowledge agents about the communication interactions of the new agents. Communication is performed by inserting and removing tuples from the tuple space. If a knowledge agent is defined to take action based upon the existence of a tuple, the data-driven nature of the database will notify the knowledge agent of the ability of the rule to fire.

In addition to the tuple space model, it is entirely possible to perform message passing and other forms of communication by defining new functions that implement message passing to the

knowledge language. Thus the tuple space, although probably the primary mechanism for communication in this model, need not be at all restrictive to robust communications.

1.2. The Model Overview

Figure 1 shows the general architecture for supporting distributed knowledge agents. The database, to support the common tuple space, is the only module common to the agents. Even then, a common database is not a requirement if alternative forms of message passing are chosen. The database interface and Knowledge Base Management System (KBMS) are instantiated once on each physical platform.

In the figure a database is shown that exists independently of the KBMS. How the database is implemented is not important to the operation of the KBMS. The database interface is concerned with interfacing knowledge agents to the database. If a tuple is asserted to the database by a knowledge agent, the database interface will both add it to the database as well as notify any other knowledge agents that use that tuple of it. Rule Group 1 represents one knowledge agent and Rule Group 2 represents another. Both of these knowledge agents happen to be defined in a single instantiation of the KBMS system. Rule Group 3 and 4 are sub-rule groups and are part of the knowledge agent consisting of Rule Group 1. Another way to think of it is that each rule group represents a distinct knowledge agent where Rule Group 3 and 4 are strictly controlled by the agent of Rule Group 1. If another knowledge agent existed on another physical platform, there would be an additional instantiation of a database interface and KBMS. The database itself would remain (whether distributed or not) the same.

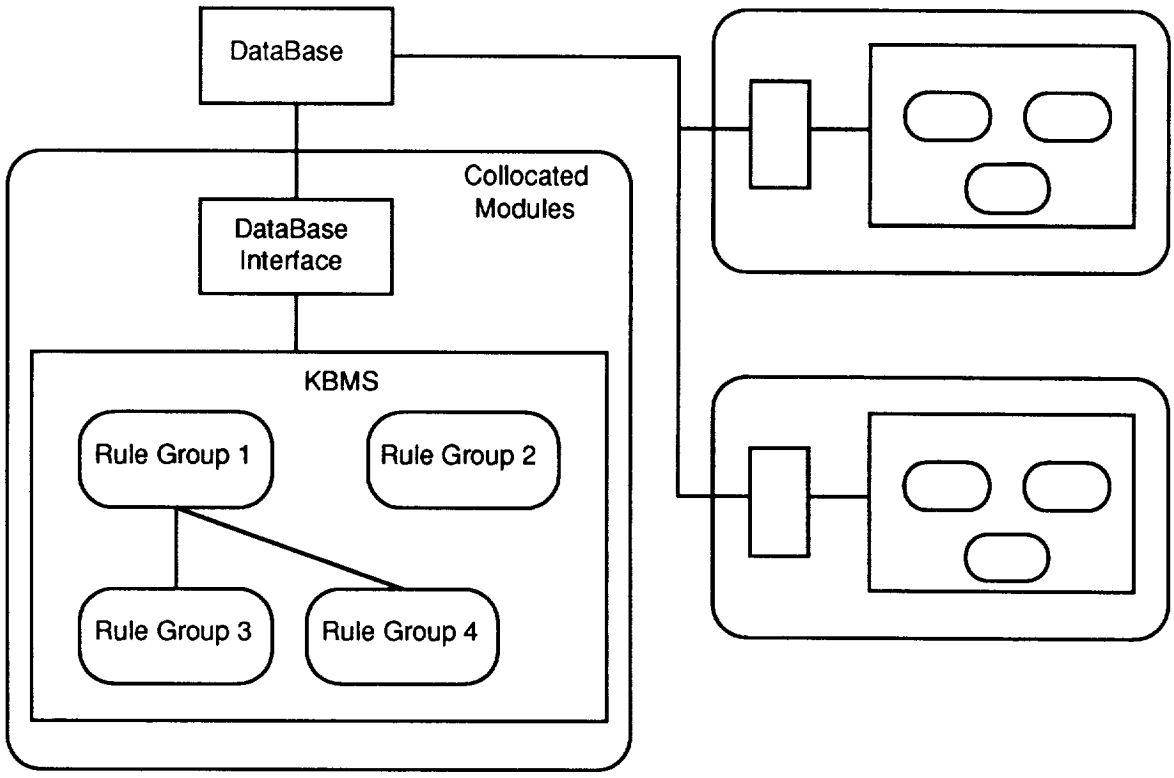

Figure 1 – KBMS Architecture

2. The Model: Its Design and Implementation

The model described here is based on a shared database, used to communicated between various knowledge agents through the passing of tuple data structures. Frames are defined on this base through a database interface to the knowledge agents, allowing data abstraction, inheritance, and structuring. Each knowledge agent contains domain knowledge in the form of rules and data and is controlled by the KBMS. The knowledge language of the KBMS provides the support to implement knowledge agents and allows arbitrary function calls. Tractability is maintained by a combination of rule compilation provided by the system and heuristic control knowledge provided by the knowledge engineer. The following subsections describe the design of the KBMS in much more detail.

2.1. The Base: The Database and Tuple Space

The first level of support for a KBMS is working memory, or the database. The architecture presented here views the database as a plug compatible module. This provides the ability to take advantage of existing databases and database management mechanisms such as ORACLE or INGRES for example.

The KBMS takes a tuple space view of the world and requires that the database represent them. A tuple is an ordered sequence of (possibly) typed fields. Conceptually, a tuple may be thought of as an object that exists independent of the process that created it. This implies that a place to store it is needed if it is to live a life independent of processes used to create and destroy it. The tuple space is sufficient for representing First Order Logic (FOL) and also sufficient for representing at least relational databases ([17]).

The tuple space has also been proposed as a mechanism for supporting distributed processing (the LINDA model, see [5]). There are three basic mechanisms that a process may perform on tuples in the tuple space: `IN`, `OUT`, and `READ`. The tuple fields in the `IN` and `READ` operations may optionally contain variable arguments for matching. `IN` and `READ` block until the tuple is present in the database. `IN` subsequently removes the tuple from the database while `READ` does not. `OUT` installs a tuple in the database. The benefit to distributed processing that the tuple space provides is the ability to add processes to the environment and have them communicate with existing processes without first having to encode knowledge about existing processes directly into their formalism. It has also been proposed that these basic tuple space operations are sufficient for supporting other communication mechanisms such as those defined in contract nets and Actors. On the other hand, the use of `IN` must be judicious in order to support the addition of other processes that may also need to use the tuple. See [5, 14] for more details.

The database provides three operations similar to the tuple space operations, these are: `STORE`, `RETRIEVE/MATCH`, and `DB-REMOVE`. The major difference between these operators and the tuple space operators `IN`, `READ`, and `OUT`, is that these do not block. Another difference is that while `IN` and `READ` will non-deterministically select one of multiple matching tuples, `MATCH` will return all of them (`RETRIEVE` will also act non-deterministically). Although the database operations defined here are not identical to the tuple space operations in the LINDA model (they do not block), the operations, in combination with the data-driven nature of a KBMS are probably sufficient for managing distributed processing protocols as LINDA does ([5]).

This view of the database allows for many different implementations of the database to be used, including distributed databases. A distributed database would provide the ability to support distributed knowledge agents defined using the knowledge representation presented here. Their

communication is then supported by the tuple space mechanism as implemented in the distributed database.

The implementation of the database for the KBMS described here also supports a form of integrity constraints and a restricted version of views. An integrity constraint is defined by a tuple. Fields of the tuple may contain one of four items: An actual field descriptor (formal argument); a variable (to match an argument); a type specifier, meaning that the field may match tuples with the corresponding field of the same type; and finally, a type declarator, meaning that for all tuples matched on the other three options, this field must be of the declared type. Views provide different databases for storage and retrieval. This is quite different to the traditional use of views in most databases. What this provides is a way to organize data into logical groupings. Currently, the use of integrity constraints is global across all views, in the future this should use the view mechanism just as any other storage or retrieval operation does.

2.2. The Next Level: Data Abstraction and Inheritance

The database interface is the module that links the database to the KBMS and vice versa. The interface provides the necessary hooks for proper accessing and notification of data in the database and of interest to the KBMS. The database interface provides the operations `STORE!`, `REMOVE!`, `MATCH!`, and `RETRIEVE!`. It also provides a data and procedure abstraction mechanism–a frame system ([10, 15]). Frames are used to extend the knowledge representation language of the KBMS for supporting complex domains requiring novel abstractions and inheritance.

Frames are an abstract organization of data into conceptual units. In this definition data may be any object, it may be simple data or even procedural objects. A frame may have any number of slots. Frames may be defined as children of multiple parents (making inheritance potentially more complex) and may also have code attached to them that is executed whenever a new instance of the frame or one of its children is created. Slots may have six optional aspects. The most used aspect is the `:value` aspect. This aspect is where a value for the slot is located. The `:if-needed` aspect is used to store code that is executed if a slot value is asked for. The `:if-added` aspect is used to store code that is executed whenever the slot gets a new value. There are two aspects that are used to constrain the value of the slot. The `:constraint` aspect is used to store code that checks if an added value passes the constraint. Since this is user-defined code there is no restriction on what it may do, only that it return a true or false status indicating the result of the constraint. The `:mustbe` aspect constrains the value to be one of a list of formal values or frames. Finally, the `:distribution` aspect is used to determine if the value of the slot is for global distribution, accessible to all knowledge agents, or only for local use.

Frame data is stored in the database as 5-tuples of the form: `(frame <name> <slot> <aspect> <value>)`. Obviously, there is no restriction on the value aspects, they can be any normal data type as well as executable code. Facts are stored in the database as 4-tuples: `(fact <name> :value <value>)`.

Frame inheritance information must also be stored in the database so that inheritance over frames may take place in response to different knowledge agent's requests for data values. The current implementation does not yet store this data in the database.

It is the responsibility of the database interface to both store and retrieve information and to notify the KBMS of changes to data made by other knowledge agents. Thus, if knowledge agent 1 makes a change to the value of a fact and knowledge agent 2 uses that fact on the left hand side (LHS) of a rule, it is the responsibility of the database interface (and distributed database) to notify knowledge agent 2 of the changed fact. And vice versa, it is the responsibility of the system, when

knowledge agents are being defined to notify the database interface of which knowledge agents index on which facts.

2.3. The Knowledge Base Management System

Now that the basic knowledge storing and retrieving mechanisms have been outlined, the heart of the system needs to be defined.

The knowledge representation language is defined as a set of objects up to the level of rules. A KBMS is then instantiated in the language using a combination of user-defined objects, user-defined code, rules and KBMS domain knowledge (i.e. knowledge about what the KBMS is). The KBMS instantiated here consists of the user-defined objects: `rule-group` and `knowledge-base`. The default inference strategy is implemented in code, but can also be implemented in the form of meta-rule groups. Rules for defining control strategy, for instance, are defined. The domain knowledge consists of frame knowledge about rule groups, knowledge bases, and knowledge of executable procedures.

To go one level further, an expert system is then instantiated from the KBMS. Here, rules and rule groups are provided for the domain; domain knowledge is provided, as well as more specific inference strategies for the expert system. To use an analogy from the `flavors` object oriented system, the knowledge representation language is like a base flavor. It is necessary to define a mixin to it in order to give it functionality. Finally, the flavor can be instantiated into a user-definable object (e.g an expert system for fault diagnosis).

The knowledge representation language provides functionality to the level of rules. A rule has the basic form of LHS ::> RHS, and may include else conditions. Quantified rules quantify over some set, binding a variable to successive values and executing the sub-rule of the quantified rule. Rules have the basic operations: `evaluate-lhs`, `evaluate-rhs`, `interpret-lhs`, `interpret-rhs`, and `interpret-else`. These operations may return one of three values: `:ok`, `:ng`, and `:missing-patterns`. If the rule evaluates `:ok` is returned, if conditions are not met `:ng` is returned, and finally, if patterns needed to check conditions do not yet exist (i.e. values do not exist for referenced variables) `:missing-patterns` is returned. These return values are used by various control strategies that can be user-defined.

The current implementation of the knowledge representation language assumes the existence of a `rule-group` definition that must include the slots: `*lhs-tickled-queue*` and `*rhs-tickled-queue*`. These slots are used to queue up rules whose LHS and RHS (respectively) are potentially `:ok`. This allows a wide variety of control strategies to be defined including forward and backward chaining as well as various combinations thereof.

The KBMS implemented here defines a knowledge base to consist of a number of rule groups as well as domain knowledge. Each rule group provides mechanisms for defining aspects of the inference strategy using either further rule groups or user-defined functions. A rule group also has a mechanism for specifying the level of determinism desired over rule execution ([1, 9]). Basically, the rule group inference strategy consists of a match, evaluate, and fire loop. The match phase is supported by the database interface which automatically queues up potential rules on the tickled-queues. The evaluate phase then checks the rules on the tickled-queues to determine which ones belong in the conflict set. Finally, one rule is selected and fired. This approach is completely non-deterministic in determining which rules get fired from the conflict set. In between the match and evaluate phases a control phase is added. This control phase consists of the definition of a regular expression that defines which rules may be fired next. In this implementation the transition table derived from the regular expression is given instead of the regular expression itself. This is much simpler from the user's perspective. For the user defining a number of rules and wanting to

insert some control over them, it is easier to specify a transition table over the rules than it is to define the regular expression that the transition table can be derived from. Furthermore, it is easier to maintain a transition table than a regular expression during rule group modification.

The knowledge representation language, although not completely independent of the KBMS, defines a very general mechanism for specifying knowledge and inference. The KBMS defined here is also very general and allows for a wide variety of specification of inference strategies over the knowledge. This can be done using the knowledge language or by alternatively writing executable code directly (i.e. interpreted vs. compiled).

2.4. The Tractability of the KBMS

Considering the generality and expressivity of the knowledge language and the KBMS, efficiency has the potential of getting lost. There are three ways of maintaining tractability in the knowledge language.

The first way of maintaining tractability is to manage the match phase of the inference cycle efficiently. The conceptual definition of the match phase is to check each rule and evaluate its LHS (RHS) and if it is `:ok` then to queue it up. Obviously, this is also the slowest approach. The step this implementation takes is to compile rules into a rule constraint network that maintains rules in a form more suited for recognizing when data becomes available that has the potential of satisfying the LHS (RHS) of a rule. Consider the LHS of a rule. During evaluation, all the referenced variables must have values in order to determine if the LHS is satisfied. The rule constraint network represents variables as nodes in the network and rules as rule-nodes (see Figure 2). As variables get values, the nodes representing the variables are triggered. All the rule-nodes connected to the node are then triggered. These rule-nodes are then checked to see if all the nodes representing the LHS (RHS) variables have a value. If so, the rule represented by the rule-node is then put on the appropriate queue. As can be seen, this method allows rules to be tickled that may not be satisfied. The rule constraint network only checks to see if variables have values, not if they have the right value. Thus the rules on the tickled queues must still be evaluated for satisfaction.

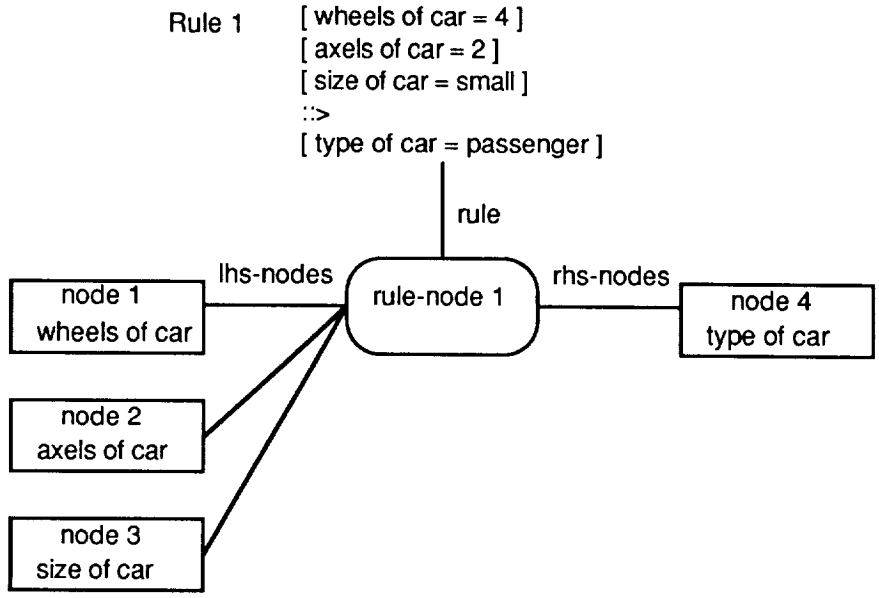

Figure 2 – Rule Constraint Network Example

Quantified rules are also compiled into this form as the variables being quantified over are given values. Thus, if a rule asks if there exists a symptom in the symptom-set and there are three symptoms in symptom-set, then three instances of this graph structure will be created, one of which may cause the rule to fire.

Alternatively, rules could be compiled more completely so that variables are checked to see if they have the right value. The complexity of the knowledge language implemented here makes this a difficult task and it has not been determined that it would be cost effective. Languages with less expressivity can be completely compiled much more easily (such as OPS5 using the RETE network [7, 8]). The expressivity and tractability trade-off turns up once again.

The second way of maintaining tractability is by providing code for inference strategies instead of providing rules defining the inference strategy. This is making use of the compiled vs. interpreted option. This means that a programmer fluent in the language that the knowledge language is implemented in needs to be available for both implementation and maintenance. However, considering the expressivity of the knowledge language, this may be a cost effective solution. It is very easy for the programmer to express what is desired without having to go into contortions over representational limits.

The third way to maintain tractability is by the effective use of knowledge. This is to make use of Raj Reddy's fifth principle: "Knowledge eliminates the need for search" ([16] see also [12]). In other words, the domain to be represented needs to be analyzed for maximum efficiency in terms of knowledge organization. For example, I can have two rule groups; one rule group forward chains on various data and computes a value for the variable `diagnosis`. The other rule group then uses the value of `diagnosis` to output the results to the user. If there are 50 rules in each rule group that use the variable `diagnosis`, then 100 rules are triggered whenever the value of the variable changes. If both of these rule groups are made to be sub-rule groups of a control rule group, the first rule group can compute a value for `diagnosis`. The control rule group can then set the value for the variable `the-diagnosis` as the value of `diagnosis`. `the-diagnosis` is then used by the second rule group instead of `diagnosis`. Now only 50 rules get triggered at one time.

The representation of knowledge should make maximum use of divide and conquer principles of knowledge organization. Another approach along the same lines is to provide strong heuristic knowledge to eliminate the need for search. This can come in many forms including domain guided inference strategies over rule groups as well as judicious use of control over the execution of rules in a rule group (i.e. the transition table method over rules).

3. A Power Management and Distribution Knowledge Agent

In this example a knowledge agent for managing power and distribution for Space Station Freedom is presented. The knowledge base consists of approximately 150 rules at this time. Naturally, space limitations prohibits the presentation of the entire knowledge base. This example should be sufficient to illustrate the representational capabilities of the knowledge language and how it can be applied to defining multiple agents for distributed processing tasks.

The first part of this example consists of the definitions required by the knowledge language to support the KBMS. The second part then describes the Power Management and Distribution Knowledge Agent.

3.1. Definitions for KBMS Support

These definitions define the user-accessible structure of a knowledge base and of a rule group. The lisp frame is for defining various functions that may be called from rules.

```
(frame :name knowledge-base
       :slots ((rule-groups :value nil)
               (agents :value nil)
               (name :value nil)))

(frame :name rule-group
        :slots ((rules :value nil)
                  (name)
                  (quantified-vars)
                  (rg-var)
                  (plan)
                  (plan-state)
                  (plan-table)
                  (viable-set :value nil)
                  (not-yet-viable-set :value nil)
                  (fire-set :value nil)
                  (local-variables)
                  (conflict-set :value nil)
                  (tickle-set :value nil)
                  (satisfied-set :value nil)
                  (unsatisfied-set :value nil)
                  (cant-fire-set :value nil)
                  (fired-set :value nil)
                  (untickled-set :value nil)
                  (*lhs-tickled-queue* :value nil)
                  (*rhs-tickled-queue* :value nil)
                  (termination-condition :value nil)
                  (control-strategy :value
                        #'default-control-strategy)
                  (conflict-resolution-strategy :value
                        #'default-conflict-resolution-strategy)
                  (execute :value #'execute)))

(frame :name lisp
        :slots ((evaluate :value #'evaluate)
                  (evaluate-lhs :value #'evaluate-lhs)
                  (evaluate-rhs :value #'evaluate-rhs)
                  (interpret-rhs :value #'interpret-rhs)
                  (interpret-lhs :value #'interpret-lhs)
                  (interpret-else :value #'interpret-else)
                  (first :value #'first)
                  (second :value #'second)
                  (format :value #'format)
                  (length :value #'length)))
```

3.2. Power Management and Distribution Knowledge Agent

The main knowledge base is defined here. It is called pmad and uses a domain file (domain.lisp) to define the various data structures (domain knowledge) relevant to the knowledge agent. The knowledge base consists of three rule groups; a control rule group for controlling the

diagnosis of hard faults, a hard fault rule group for computing a diagnosis, and a diagnosis rule group simply for printing a diagnosis.

```
KB : pmad

DOMAIN : domain.lisp

RULE-GROUP : control-rg
```

This control rule group currently manages the collection of fault information for diagnosis. It controls the execution of two rule groups: hard-fault and diagnosis, for performing and printing out diagnosis information respectively. Eventually rules will be added to this knowledge agent for soft fault and incipient fault analysis.

```
CONTROL : ((start (Control-Rule1))
           (Control-Rule1 (Control-Rule2))
           (Control-Rule2 (Control-Rule3 Control-Rule4
                           Control-Rule5))
           (Control-Rule3 (Control-Rule3 Control-Rule4
                           Control-Rule5))
           (Control-Rule4 (Control-Rule3 Control-Rule4
                           Control-Rule5))
           (Control-Rule5 (Control-Rule6))
           (Control-Rule6 (Control-Rule1)))


Control-Rule1
THERE EXISTS symptom-set in symptom-set-queue
  < ::>
    [ the-symptom-set = symptom-set ]
    [ diagnosis-set = empty ] >
;
Control-Rule2
::>
[ clusters = power-domain :: cluster-symptoms
                          ( the-symptom-set ) ]
;
Control-Rule3
THERE EXISTS cluster in clusters
  < ::>
    [ symptoms of symptom-set1 = symptoms of cluster ]
    [ symptom-set1 = cluster ] >
  ELSE
  [ ready-to-diagnose = false ]
;
Control-Rule4
THERE EXISTS symptom in symptoms of symptom-set1
  < [ hard-fault :: execute ( hard-fault ) = :ok ]
    ::>
    [ diagnosis-set = diagnosis-set PLUS diagnosis ]
    [ the-diagnosis = diagnosis ]
    [ diagnosis = :unknown ]
    [ diagnosis-rg :: execute ( diagnosis-rg ) ]
    [ clusters = clusters MINUS symptom-set1 ] >
;
Control-Rule5
[ ready-to-diagnose = false ]
[ FOR ALL diagnosis1 in diagnosis-set
  < [ diagnosis1 = diagnosis-2 ]
```

```
      OR
      [ diagnosis1 = diagnosis-31 ] > ]
  ::>
  [ the-diagnosis = diagnosis-no-power ]
  [ diagnosis-set = diagnosis-set PLUS the-diagnosis ]
  [ ready-to-diagnose = true ]
  ELSE
  [ ready-to-diagnose = true ]
;
  Control-Rule6
  [ ready-to-diagnose = true ]
  ::>
  [ diagnosis-rg :: execute ( diagnosis-rg ) ]
  [ symptom-set-queue = symptom-set-queue
                              MINUS the-symptom-set ]
  [ symptom-set1 = :unknown ]
  [ clusters = :unknown ]
  [ ready-to-diagnose = :unknown ]
:

RULE-GROUP : hard-fault.rg
```

This rule group takes information about a fault and, after determining if any testing needs to be done and doing it, determines the fault and assigns diagnosis a value.

As can be seen, the control that is specifiable over a rule group may be quite complex. The whole transition table for this rule group is not given (although this is one-third of it, for a rule group consisting of about 90 rules), but the expressivity is still quite apparent.

```
  CONTROL : ((start (init-rule))
               (init-rule (Rule1))
               (Rule1 (Rule2))
               (Rule2 (Rule3 Rule4.1 Rule5 Rule31 Rule31.1
                             Rule31.2 Rule35 Rule35.1))
               (Rule3 (Rule4))
               (Rule4 (Rule20 Rule32))
               (Rule32 (Rule33))
               (Rule33 (Rule34))
               (Rule4.1 (Rule4.2))
               (Rule4.2 (Rule4.3))
               (Rule4.3 (Rule4.4 Rule4.5 Rule4.6 Rule4.7
                             Rule4.8))
               ...
              )

  Init-rule
  ::>
  [ symptom-set = symptoms of symptom-set1 ]
  [ possible-top-switches = empty ]
  [ tripped-top-switches = empty ]
;
  Rule1
  ::>
  [ top-symptoms = power-domain ::
                        top-symptoms ( symptom-set ) ]
;
  Rule2
  [ lisp :: length ( top-symptoms )  > 1 ]
```

```
 ::>
 [ type = multiple-tops ]
 ELSE
 [ type = single-top ]
;
 Rule3
 [ type = multiple-tops ]
 [ THERE EXISTS symptom in top-symptoms
   < [ lisp :: length
           ( switches-below of switch of symptom ) > 0 ] > ]
 [ FOR ALL symptom in top-symptoms
   < [ fault of symptom = fast-trip ]
     OR
     [ fault of symptom = over-current ] > ]
 ::>
 [ FOR ALL symptom in top-symptoms
   < [ tripped-top-switches = tripped-top-switches PLUS
                                        switch of symptom ] > ]
 [ THERE EXISTS symptom in top-symptoms
   < [ possible-top-switches = switch of symptom PLUS
                               siblings of switch of symptom ] > ]
 [ THERE EXISTS symptom in top-symptoms
   < [ trip-type = fault of symptom ] > ]
 [ type = multiple-top-current-trip ]
;
 ...
 Rule4.4
 [ type = multiple-tops ]
 [ lisp :: length ( new-symptoms ) = 1 ]
 [ THERE EXISTS symptom in new-symptoms
   < [ THERE EXISTS symptom1 in top-symptoms
       < [ switch of symptom = switch of symptom1 ]
         [ fault of symptom = fault of symptom1 ] > ] > ]
 ::>
 [ diagnosis = diagnosis-54 ]
;
 ...
:
 [ diagnosis ]

RULE-GROUP : diagnosis.rg
```

This rule group simply prints out some statements and communicates diagnostic information to the scheduling knowledge agent depending on the particular diagnosis encountered.

```
  d-rule1 @@ backrush in load center
  [ the-diagnosis = diagnosis-54 ]
  ::>
  [ the-diagnosis = :unknown ]
  [ lisp :: format ( t "~%The following load center
                 RPCs tripped on fast-trip~%" ) ]
  ...
;
  ...
:
  [ the-diagnosis = :unknown ]
```

The rest of the knowledge base for this knowledge agent.

```
Domain-Knowledge :
  constants :
    t ; :ok ; :ng ; :missing-patterns ; yes ; no ;
    hard-fault ;
    diagnosis-rg ;
    multiple-tops ;
    single-top ;
    multiple-top-current-trip ;
    over-current ;
    under-voltage ;
    fast-trip ;
    ground-fault ;
    diagnosis-1 ;
    diagnosis-2 ;
    ...
    diagnosis-no-power .
  facts :
    empty = ( ) .
  frames :
    (fcreate-instance 'symptom-set 'symptom-set1) .

Begin : control-rg

END-KB


++++++++++++++++
domain.lisp
++++++++++++++++

(frame :name power-domain
       :slots ((top-symptoms :value #'top-symptoms)
               ...
               (close-switch :value #'close-switch)))

(frame :name symptom-set
       :slots ((symptoms)))

(frame :name symptom
       :slots ((switch) (fault)))

(frame :name switch
       :slots ((name)
               (type)
               (current)
               (switches-below :value nil)
               (switch-above :value nil)
               (siblings)
               ...
               (current-rating)
               (fast-trip-percent)))

...
```

Lots of domain knowledge here to build instances of switches and sensors, etc. Knowledge of the topology is encoded here. About 30k worth.

4. Conclusions and Future Work

An architecture for defining and modifying knowledge base management systems that may be used for applications in distributed AI has been presented. The architecture is very flexible and relatively efficient. It has been used to define three very different knowledge agents: One for solving a toy problem to compute how to send a package consisting of about ten rules; One for solving the monkeys and bananas problem consisting of about twenty fairly complex rules, this one was directly adapted from a solution given for OPS5; Finally the agent given in this paper and consisting of around 150 rules.

The results we have noticed so far have shown that this architecture provided the ability to easily implement a solution to a wide variety of problems. The monkeys and bananas problem has driven out many areas of weakness in the implementation that are being dealt with. The speed with which this architecture solves the monkeys and bananas problem is hardly even comparable to that of OPS5 at this point. However, the result of implementing our fault diagnosis problem for power management and distribution has turned out very well. Using simple forward chaining and lots of control knowledge in the hard fault rule group has enabled us to provide a solution that is very fast and easily maintainable. The maintainability is very important for this domain as the requirements for Space Station Freedom have not been completely specified.

4.1. Future Work

This system is being implemented as part of a much larger system, KNOMAD (Knowledge Management and Design System). We have identified a number of areas where knowledge needs to be added to support a completely robust, domain independent environment for specifying knowledge based systems. These include the addition of a constraint system, a temporal database, and analytical and qualitative reasoning. These additions will then support planning, scheduling, and causal reasoning at the least. Adding these components must involve how the KBMS will access and use these components as well as how these components will use the existing database and database interface mechanisms.

Work also needs to be pursued to determine the possibility of adding RETE-like structures as a part of the rule constraint network.

5. Acknowledgements

This work is being supported by NASA, Marshall Space Flight Center, contract NAS8-36433.

6. References

[1] Baskin, A.B., "Combining Deterministic and Non-deterministic Rule Scheduling in an Expert System," AAMSI, 1986.

[2] Bond, Alan H., and Les Gasser, Eds., "Readings in Distributed Artificial Intelligence," Morgan Kaufman, 1988.

[3] Brodie, M.L., J. Mylopoulos, and J.W. Schmidt, (Eds.), "On Conceptual Modelling," Springer-Verlag, New York, 1984.

[4] Buchanan, Bruce G., and Richard O. Duda, "Principles of Rule-Based Expert systems," Stanford Heuristic Programming Project Report No. HPP-82-14, 1982.

[5] Carriero, Nicholas, and David Gelernter, "Linda in Context," Communications of the ACM, 32(4), 1989.

[6] D'Angelo, Antonio, Giovanni Guida, Maurixo Pighin, and Carlo Tasso, "A Mechanism for Representing and Using Meta-Knowledge in Rule-Based Systems," Approximate Reasoning in Expert Systems, 1985.

[7] Forgy, Charles L., "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," AI 19(1) 1982.

[8] Forgy, Charles L., and Susan J. Shepard, "RETE: A Fast Match Algorithm," AI Expert, Jan. 1987.

[9] Georgeff, M.P., "Procedural Control in Production Systems," AI 18, pp. 175-201, 1982.

[10] Hayes, P.J., "The Logic of Frames," in Webber, B.L., and Nils J. Nilsson (Eds.) *Readings in Artificial Intelligence*, pp. 451-458, 1981.

[11] Hayes-Roth, Frederick, "Towards Benchmarks for Knowledge Systems and Their Implications for Data Engineering," IEEE Transactions on Knowledge And Data Engineering, Vol. 1, No. 1, March 1989.

[12] Lenat, Douglas B., and Edward A. Feigenbaum, "On the Thresholds of Knowledge," International Workshop on Artificial Intelligence for Industrial Applications, 1988.

[13] Levesque, Hector J., "Knowledge Representation and Reasoning," Annual Reviews of Computer Science, 1986.

[14] Matsuoka, Satoshi, and Satoru Kawai, "Using Tuple Space Communication in Distributed Object-Oriented Languages," OOPSLA 1988 Proceedings.

[15] Minsky, M., "A Framework for Representing Knowledge," in P. Winston (Ed.) *The Psychology of Computer Vision*, McGraw-Hill, New York, pp. 211-277, 1975.

[16] Reddy, Raj, Presidential Address at the American Association for Artificial Intelligence Conference, 1986.

[17] Reiter, R., "Towards a Logical Reconstruction of Relational Database Theory," in [3], pp. 191-233, 1984.

[18] Stefik, Mark, Jan Aikins, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, and Earl Sacerdoti, "The Organization of Expert Systems, A Tutorial," AI 18, 1982.

[19] Wilkins, David E., "Practical Planning: Extending the Classical AI Planning Paradigm," Morgan Kaufman, 1988.