FrameUP: A frames formalism for expert systems

Graham J. Williams
Department of Computer Science, Australian National University, Canberra, Australia.

This paper presents an introduction to frames-based representation schemes for use in the construction of rule-based expert systems. The features that are relevant to such expert systems are discussed, followed by an example of the type of rule application mechanism that the system implements. Advantages of such a system are presented.

Keywords: Knowledge representation, frames, expert systems, artificial intelligence, FrameUP.
Categories: I.2.1, I.2.4.

Manuscript received December 1986; revised February 1988.

1. INTRODUCTION

A **knowledge-based system** is a computer system which employs a knowledge base and an inferencing mechanism to solve problems. An important aspect of such systems is that the knowledge base is maintained separately from the inferencing mechanism. The rationale for such separation is that the usually declarative knowledge contained in a knowledge base can be more easily modified and updated if there is no need to refer to the inferencing machinery. Such systems are also identified by their ability to explain the steps taken in solving a problem, by reference to the knowledge base. Also, the knowledge base is usually represented in a readily human-understandable form.

A **rule-based system** is a knowledge-based system in which some variation of the production rule (Davis and King, 1984) is used to encode knowledge. The term "expert system" has been used to refer to knowledge-based systems which perform tasks usually performed by "experts". The vast majority of these expert systems have in fact been rule-based systems (Waterman, 1986). A number of deficiencies however have arisen with the use of rule-based systems (Williams, 1986). These include restrictions imposed by the knowledge representation scheme, the encoding of much implicit knowledge, the knowledge acquisition bottleneck, and the inability to add in new types of knowledge to the system after the system has been developed. Richer methods for representing knowledge for use by expert systems have thus been sought. It is in this context that the frames formalism is introduced.

FrameUP is a representation system based upon frames. It was developed specifically to provide the facilities required for the implementation of an expert system. The Household Electrics Fault Finding Expert (HEFFE) has been implemented using FrameUP. This expert system has the task of determining the cause of faults in household electrical wiring.

The aim of the work described here is to represent all elements of an expert system (tasks, rules, procedures, etc.) within the uniform structure provided by frames. We show that at least some of the problems of expert systems, as discussed in Williams (1986), can be solved. The aim of this paper is to introduce the terminology of Frames and to illustrate the use of frames for implementing an expert system.

2. FRAMES

A **frame** is a named node of a hierarchical network representing a concept or object. Associated with each frame is a collection of **slot-value** structures which represent the various attributes of the concept or object being represented. Frames are thus somewhat like record structures as found in other programming languages, where the fields of a record correspond to the slots of a frame. However, the values of the slots may be procedures, symbols (numbers, strings), or the names of other frames, and we may have one or more of these associated with a single slot at any one time. A distinguishing feature of frames is that when values are added or removed from slots, procedures may automatically be activated to carry out a variety of other tasks.

The frames scheme was proposed as a way of representing stereotypical situations (Minsky, 1975), where a stereotype of a class of objects is a description of those features (slots) expected to be found associated with each object of that class. Some of these slots might represent defaults, and are used, in the absence of other information, to provide values for particular instances of the class of objects described by the stereotype. Other slots record the distinguishing characteristics of the class of objects: characteristics which each object of the class will inherit.

Figure 1 illustrates a simple frame from the world of household electrical items. This frame represents an instance of the class of objects known as DeskLamps, and is named **DeskLamp1**. The other slots indicate that this particular DeskLamp is NotOperating, that it has the relation *PluggedInto* with the object named PowerPoint1, and that it *OperatesFrom* an object named PowerPoint2. DeskLamp, PowerPoint1 and PowerPoint2 are in fact the names of other frames. The *IsA* slot defines the position of this frame in the hierarchy.

For clarity, the terminology **frame.***slot* is introduced. Thus, instead of saying that DeskLamp is the value of the *IsA* slot of the frame for **DeskLamp1**, we say that DeskLamp is the value of **DeskLamp1**.*IsA*, or more concisely, that **DeskLamp1**.*IsA* DeskLamp.

```
DeskLamp1
    IsA: DeskLamp
    Status: NotOperating
    PluggedInto: PowerPoint1
    OperatesFrom: PowerPoint2
```

Figure 1. A frame representing an object.

3. LISP TERMINOLOGY

FrameUP is implemented in a version of the LISP programming language called Franz LISP. A basic understanding of LISP is required in order to follow the discussion presented below. A brief introduction to the necessary concepts follows.

LISP is a list processing, functional programming language. The basic data type is a list, which has the form (a b c d), where a, b, c and d are either atoms (names) or are themselves lists. An empty list () is denoted by the atom nil. A list consists of a head and a tail. In LISP terminology, these are the car and the cdr (pronounced "could-er"). The car (head) of the list above is the atom a, whilst the cdr (tail) is the list (b c d). When the cdr of the list is an atom, a dotted pair is used to represent the list. Thus, the list (a . b) has the atom a as its car and the atom b as its cdr, whereas (a b) has a as its car and the list (b) as its cdr.

The concept of a function is central to LISP — a function takes a number of arguments and returns some value. A function definition is identified in LISP as a list whose car is the special symbol lambda. The cdr consists of a list of arguments followed by the definition of the function. Functions can be associated with an atom, thus giving a name to the function. Function application is effected by having a function or the name of a function as the first member of a list, with the rest of the list being the arguments of the function. The following example illustrates both a function definition, and function application. It defines a function which returns the average of its two arguments, x and y. The body of the function uses the two LISP built-in functions called sum and quotient, and illustrates how the result of one function application can be used as the argument for another. LISP uses the first-most inner-most rule of evaluation, so that the sum of x and y is calculated, and then it is divided by 2.

```
(lambda (x y) (quotient (sum x y) 2))
```

Predicates in LISP are functions which return nil to indicate falsity, and t or any other non-nil object to indicate truth. The predicate or, for example, evaluates to t if at least one of its arguments is non-nil. Otherwise it returns nil. Similarly, (member x y) is true if x is a member of the list y, and (equal x y) is true if x and y are the same objects.

Variables in LISP are atoms with values associated (or 'bound') to them. Also associated with each atom we can have a property list. This provides a place to store data associated with the atom. Each property has a name and an associated value. LISP provides functions to store and to access these values — (putprop 'a 'b 'c) stores the value b as property c of a, and (get 'a 'c) returns the value stored as property c of a (i.e. b). The quotes in these LISP expressions indicate that what follows is to be treated much like a constant — it is not to be evaluated.

Similar in concept to the property list is the association list. This is a list made up of sub-lists. Each sub-list has the form (atom value) where value may be an atom or a list. The value is said to be associated with the atom. The function assoc takes two arguments, an atom and an association list, and returns the value associated with the specified atom in the association list. For example:

```
(assoc 'c '((a 1) (b 2) (c 3) (d 4)))
```

returns (3).

A variable may have bound to it a list which has the form of a function (i.e. its car is lambda etc.), or a function can return a list whose form is that of a function. We can apply such functions to arguments with the LISP apply. For example, if the list

```
(lambda (x y) (quotient (sum x y) 2))
```

is bound to the atom a then (apply a '(1 3)) returns 2, the average of 1 and 3.

Finally, LISP also provides a macro facility. One predefined macro in Franz LISP is called let. This has the form:

```
(let ((x = a) (y = b) ...) lisp-expression)
```

where all occurrences of x and y in the lisp-expression are replaced by the values of a and b respectively.

4. A FRAMES REPRESENTATION LANGUAGE — RLL

Greiner and Lenat (1980) developed the frames idea into a representation language (RLL), in which as much as possible has a representation using frames. RLL is described as an "expert system whose 'task domain' is itself the building of expert systems" (Barstow et al, 1983).

RLL incorporates many of the usual features of a frame scheme. In particular, objects, concepts and generic classes are represented as frames, and an inheritance mechanism is provided. But an important new idea introduced in RLL is that for each type of slot known to the system there should be a frame representing information about that slot. Figure 2 illustrates such a frame for the slot *PluggedInto* which appeared in Figure 1. The information recorded about a slot includes the type of values the slot can have (specified by the *Format* and the *DataType*), how to compute values for the slot (*ToCompute*), and whether such computed values are to be physically stored on the slot, or computed each time as required (*Cache*). Such frames will be referred to in this paper as *slot-frames*.

```
PluggedInto
    IsA: Slot
    Description: That which this object is
                 electrically connected to.
    Format: Singleton
    DataType: Atom
    ToCompute: (lambda (fr) (or (use-rules fr)
                                (ask-user fr)))
    Cache: Never
```

Figure 2. A frame representing/defining a slot.

The frame of Figure 2 indicates that *PluggedInto* can appear as a slot on any frame. The value associated with such a slot will name another object (frame) into which the object is plugged (i.e. electrically connected). The slot *PluggedInto* can have only a single associated value which must be an atom. Whenever a value is required for the slot, the default procedure is to use the inferencing mechanism (use-rules), or if this fails, to ask the user. Also, the value associated with this slot should not be stored (*Cache*=-Never) but rather, computed each time it is required, indicating that the value may change often.

RLL defines three frame access functions: get-value, put-value and delete-value. Associated with each slot-frame **S**, either explicitly or implicitly (through inheritance), are three corresponding access-slots called *ToGetValue, ToPutValue* and *ToDeleteValue*. The function get-value uses the value stored in the slot *ToGetValue* of the frame for a specified slot whenever a value is required for that slot. Similarly, put-value and delete-value access the relevant *ToPutValue* and *ToDeleteValue* slot for instructions of how to store or to delete a value.

The value stored in any of these slots can be arbitrary functions, which may specify, for example, how inheritance is to be used, or, in the case of deleting values, which other dependent values need to be deleted. They also allow statistics relating to the operation of the system to be maintained; for example, the number of times that a frame has been accessed can be recorded. If any of these access-slots is not present, then IsA-inheritance is used to obtain a value for it. Thus, frames higher up in the hierarchy can provide these values, allowing the use of defaults.

5. FRAMEUP

FrameUP is an implementation of RLL which incorporates only those features of RLL which were found to be appropriate for the implementation of expert systems. Further, FrameUP provides explicit control strategies for use in expert systems. It implements a formalism in which tasks are the basic control structure, and production rules are the basis of inferencing. FrameUP introduces a mechanism and a control regime for the testing and execution of these production rules. Below we detail the LISP implementation of the FrameUP system. Then we deal with those aspects of FrameUP directly related to expert system construction.

5.1 Frame Representation

In FrameUP a frame is represented as a LISP list, whose car is the name of the frame, and whose cdr is a list of associations. This association list connects slot names with the values stored in the slot. The frame is stored as the property 'frame' of the atom which names the frame. An example of the representation of a frame is the list:

```
(DeskLamp1 (IsA . DeskLamp)
           (Status NotOperating)
           (Any (a b) (c d)))
```

which is stored as the property 'frame' on the atom DeskLamp1. This frame is presented pictorially here as:

DeskLamp1
IsA: DeskLamp
Status: (NotOperating)
Any: ((a b) (c d))

From this it is apparent that the slot *IsA* takes as its value a single atom. *Status* appears to take a list of atoms as its value and *Any* takes a list of lists. The formal specifications of the format and data type of these slots is stored in the corresponding slot-frames:

```
(IsA (IsA . Slot) (Format . Singleton)
   (DataType . Atom))
(Status (IsA . Slot) (Format . List)
   (DataType . Atom))
(Any (IsA . Slot) (Format . List) (DataType . List))
```

The *Format* slot of these slot-frames specify the number of values that the slot can have: "Singleton" indicates that the slot can have only one value; "List" indicates that the slot can have any number of values. The *DataType* slot specifies what type of values can be stored in the slot. The value of the *DataType* slot is the name of a frame which has a slot called Predicate having as its value a LISP predicate function. This function takes one argument and returns t if the argument is of the correct type, and nil otherwise. For example, the value of **Atom**.*Predicate* might be (lambda (x) (atom x)), which simply uses LISP's build-in atom predicate. The *Format* and *DataType* slots are used to ensure the correct storage of values in slots.

5.2 Basic Access Functions

Three FrameUP functions are defined to provide the base level access to frames. These are fup:lookup, fup:store and fup:remove and would not normally be used by the end user of the system.

The primitive retrieval function, fup:lookup takes two arguments, an atom naming a frame and an atom naming a slot, and returns the value of:

(cdr (assoc *slot* (cdr (get *frame* 'frame))))

That is, from the property list of the atom named *frame*, get the property called frame: (get *frame* 'frame). Find in the cdr of this list the appropriate *slot* and return the associated value, which is the cdr of the list returned by assoc. We refer to this returned value as the value of **frame**.*slot*. This will be nil if either the frame has not been defined, or the slot does not appear on this frame. The value nil cannot be stored as the value of a slot.

The function fup:store provides the basic storage facility. It takes three arguments, being an atom naming a frame, an atom naming a slot and the value to be stored in **frame**.*slot*. If the frame does not exist, then it is created, and the slot likewise. Use is made of **slot**.*Format* in determining if the value can be stored. If the Format is "Singleton" and a value is already stored in **frame**.*slot*, then the new value simply replaces the old value. If the Format is "List" and the new value is already a member of **frame**.*slot*, nil is returned; repeated values not being allowed. Use is also made of **slot**.*DataType* to check the data type of the item being added. The value returned by fup:store is the new value of **frame**.*slot* or nil if the attempt failed as above.

The function fup:remove can take one, two or three arguments, the first being an atom naming a frame, the second an atom naming a slot and the third a value. If only the frame name is given, then that frame is removed from the system. If a frame name and a slot name are given, then that slot is removed from the frame, along with all its values. If a value is also given, then just that value is removed. Nil is returned if the frame, slot, or value do not exist. Otherwise, the last argument in the argument list (i.e. either frame name, slot name or the value) is returned.

5.3 User Access Functions

The top-level functions defined for frame access are get-value and put-value. Their LISP definitions are given below. These two functions use information stored on the slot-frame for the given slot to determine how the slot is to be accessed. In particular, get-value uses **slot**.*ToGetValue* and put-value uses **slot**.*ToPutValue*.

The definitions are straight forward, except for the use of get-access-fun instead of get-value, in the definition of get-value. The function get-access-fn firstly attempts to use fup:lookup to simply retrieve a value from **fr**.*sl*. If this fails, the IsA-hierarchy is traversed until a value is found; default functions are always stored at the root of the hierarchy being traversed.

The concept of inheritance is used widely in frame-based systems. Any frame may inherit slot values from other frames above it in the hierarchy. In FrameUP, such inheritance can be finely controlled through the use of the LISP functions stored in the *ToGetValue* and *ToPutValue* slots. Indeed, values for these slots themselves may be inherited through the slot-frame hierarchy. A default storage function and a default retrieval function are associated with these two slots, and are stored as the values of the *ToGetValue* and *ToPutValue* slots of the frame at the root of the hierarchy of slot-frames. These default functions carry out the standard steps to be taken in storing or in retrieving a value.

The default storage function is default-put-value. This function takes three arguments; an atom naming a frame, an atom naming a slot, and a value. The first step performed by default-put-value is to check if the value argument is of the correct type. This is done by applying the predicate stored in *Slot.DataType.Predicate* to the supplied value. (The notation *A.B.C* refers to the slotnamed C which appears on the frame which is named by the value of **A**.*B*.) If the value is of the correct type, then fup:store is used to store it.

The default retrieval function is default-get-value, which takes, as arguments, two atoms, naming a frame and a slot, and returns the value of **frame**.*slot*. The value so returned may have been determined in one of a number of ways. Firstly, default-get-value uses fup:lookup to see if a value is stored for the specified slot. If this straight forward lookup succeeds, then that value is returned. Otherwise the result of (get-value slot 'ToCompute), which is expected to be a function, is applied to the atom naming the frame. The value so computed is returned as the value of the function default-get-value.

If a non-null value is found by default-get-value, then a caching mechanism may be employed. **Caching** is the storing of values so that they need not be computed each time they are required (Lenat, Hayes-Roth and Klahr, 1983). FrameUP allows caching to be controlled dynami

```
get-value = (lambda (fr sl)
                    (apply (get-access-fn sl 'ToGetValue) fr sl))
put-value = (lambda (fr sl va)
                    (apply (get-value sl 'ToPutValue) fr sl va))
get-access-fn =
        (lambda (fr sl)
                (or (fup:lookup fr sl)
                    (get-access-fn (fup:lookup fr 'IsA) sl)))
```

Figure 3. Definition of get-value **and** put-value.

cally, and specified individually for each type of slot. The function cachep is defined in FrameUP as a predicate which takes two arguments; the first an atom naming a frame, the second an atom naming a slot. It returns a non-null value if caching is to be carried out for any new values determined for **frame**.*slot*, using the value of **slot**.*Cache* to determine this. It does this as follows: Suppose the value of (get-value slot 'Cache) is X; then cachep will return t if X is the atom Always, or nil if X is the atom Never. If X is neither, then it is assumed to be a predicate function which is applied to **frame**. Cachep will return the result of this application. For a given frame and slot, if cachep returns a non-nil value, then default-get-value will put-value the computed value in **frame**.*slot*.

Consider the example of (get-value DeskLamp1 PluggedInto). Looking at Figure 2 shows that no *ToGetValue* is specified and so the default will be used — default-get-value — which is inherited from the frame illustrated in Figure 4 below (since **PluggedInto** is a Slot). Default-get-value's straightforward lookup of **DeskLamp1**.*PluggedInto* fails, so the function stored in **PluggedInto**.*ToCompute* is applied to DeskLamp1. The result of this application, if non-null is returned as the value of **DeskLamp1**.*PluggedInto*.

```
Slot
    IsA: GenericFrame
    Cache: Never
    ToGetValue: default-get-value
    ToPutValue: default-put-value
```

Figure 4. The generic slot.

In general, the *ToCompute* slot may specify that a value can be computed by some formula, or obtained from some database, or that the user is to be asked for a value, etc. This allows a great deal of dynamic flexibility in how values are to be determined.

5.4 Tasks: The Basic Control Structure

FrameUP, like RLL, employs **tasks** to represent units of action. The tasks are queued, waiting in an **agenda** until they are called upon to carry out the action they represent. These actions define the top level functionality of the system. The tasks typically deal with some slot of some frame, and may require the system to check whether the slot has a particular value, or to attempt to determine a value for the slot. In so doing, sub-tasks may be set up and placed on the agenda. The parent task will be completed only after all sub-tasks have completed.

The structure of a *task-frame* is illustrated in Figure 5. The frame and slot upon which this task is to act is identified by the *RelevantFrame* and *RelevantSlot* slots of the task-frame; in this case DeskLamp1 and ReasonForNotOperating respectively. Task1 is required to find a value for **DeskLamp1**.*ReasonForNotOperating*, i.e. to fill the slot.

For each type of task known to the system, of which FillSlot is one example, there is an associated frame which defines the steps required to effect the task. This task

```
Task1
    IsA: Task
    TypeOfTask: FillSlot
    RelevantFrame: DeskLamp1
    RelevantSlot: ReasonForNotOperating
```

Figure 5. A typical example of a top-level task for HEFFE.

definition will be in terms of the primitives: get-value and put-value. The *Procedure* slot of Figure 6 depicts the type of definition used to implement a task in FrameUP. In carrying out a task of this type, most of the work is done by get-value, so that, for example, Task1 involves evaluating (get-value DeskLamp1 ReasonForNotOperating). The result of applying the function stored on the *Procedure* slot will be a value, which will be stored on the given slot if that slot definition specifies caching.

```
FillSlot
    IsA: TypeOfTask
    Procedure: (lambda (task)
                   (get-value
                       (get-value task 'RelevantFrame)
                       (get-value task 'RelevantSlot)))
```

Figure 6. A frame representing a type of task which HEFFE can carry out.

FrameUP provides three types of tasks: FillSlot, CheckSlot and FillSlotWithGivenValue (see Figure 7 for the latter two). These have been found to be sufficient for the implementation of an expert system. A task of type CheckSlot compares the value of a given slot of a given frame with a given value; the compared value and the given frame and slot are all specified by the task. A task of type FillSlotWithGivenValue simply calls put-value to store a value for a slot and frame; which are specified by the task.

```
CheckSlot
    IsA: TypeOfTask
    Procedure: (lambda (task)
                   (let
                     ((Fr = (get-value task 'RelevantFrame))
                      (Sl = (get-value task 'RelevantSlot))
                      (Aval = (get-value Fr Sl))
                      (Rval = (get-value task 'RequiredSlotValue)))
                     (or (equal Rval Aval) (member Rval Aval))))

FillSlotWithGivenValue
    IsA: TypeOfTask
    Procedure: (lambda (task)
                   (put-value (get-value task 'RelevantFrame)
                              (get-value task 'RelevantSlot)
                              (get-value task 'GivenSlotValue)))
```

Figure 7. Task types.

The agenda, also implemented as a frame, provides a control mechanism for scheduling tasks. Its function is to order the tasks that are placed upon it and, once one task is

finished, to provide another task for execution. The ordering of tasks on the agenda can be specified as a function of, for example, an estimate of the amount of time required for the task, the importance of the task, and the other tasks on the agenda. In HEFFE, for example, the agenda simply stores tasks in the order in which they are placed on the agenda, but with sub-tasks of a current task being placed ahead of other tasks.

5.5 Rules: Their Representation and Use

Rules are used in FrameUP to carry out inferencing. A rule is represented as a frame having an if-slot and a then-slot, amongst others. The individual conditions of the if-slot and the individual conclusions of the then-slot are of the form [frame slot value], and might be compared with the associative triplets of MYCIN, which have the form (object attribute value) (Barr and Feigenbaum, 1982). Variables are denoted as atoms beginning with a '$'.

Figure 8 illustrates a rule, which is read as: "If an object (denoted by the variable $Object), which is not operating (this is implicit in the rule, but made explicit by the context in which the rule is used), is known to be sound (i.e. functionally OK) and currently connected to some power supply, then pass the blame for the object not operating on to the power supply, which apparently is not operating". The semantics of this rule in FrameUP are as follows: If Sound is a value for **$Object**.*Status* and *$PowerSupply* is the value of **$Object**.*PluggedInto,* then the new values $PowerSupply and NotOperating can be deduced for **$Object**.*ReasonForNotOperating* and **$PowerSupply**.*Status* respectively.

```
Rule1
    IsA: Rule
    If: (($Object Status Sound)
        ($Object PluggedInto $PowerSupply))
    Then: (($Object ReasonForNotOperating $PowerSupply)
          ($PowerSupply Status NotOperating))
```

Figure 8. An example rule from HEFFE to deduce why some object is not operating.

Rules are often written in terms of variables, which need to be bound to actual objects. Binding can occur at two different stages, depending upon the context in which the rule was called upon. To illustrate, consider the current context to be the task of finding a value for **DeskLamp1**.*ReasonForNotOperating.* Rule1 is known to conclude a value for this slot for any $Object. Thus Rule1 will be invoked with $Object bound toDeskLamp1. $PowerSupply will be bound to PowerPoint1 when the conditions of this rule are evaluated. A stack of bindings is used to allow for the recursive usage of a rule.

Formally, the triplet [fr sl va] appearing as a condition on a rule has the semantics that if va is a variable then that variable will be bound to the value of (get-value fr sl); otherwise va is required to be the, or a (depending upon the *Format* of **sl**), value of (get-value fr sl). If this is not the case then the condition is said to fail, and the rule in turn cannot be used in the current situation.

Triplets appearing in the condition of a rule will lead to a new task of type CheckSlot being created when the rule is called upon. These tasks will be executed in turn, unless one of them fails, in which case the remaining tasks associated with this rule are removed from consideration. These associated tasks are identified by the value of one of their slots (the *ParentTask* slot).

A triplet appearing as a conclusion of a rule has the semantics of (put-value fr sl va). All variables at this stage should have bindings. Each triplet in the conclusion leads to a new task being created, the task type being FillSlotWithGivenValue.

A rule is referenced by those slot-frames which are named as slots in the conclusion triplets of the rule. The above rule, for example, has two conclusions, naming the slots *ReasonForNotOperating* and *Status.* Thus, the slot-frames **ReasonForNotOperating** and **Status** will list Rule1 under a slot named *Rules.* The rule will be called upon whenever the *ToCompute* slot specifies that the FrameUP function use-rules is to be used.

The function use-rules takes two arguments, an atom naming a frame and an atom naming a slot. It accesses the list of rules stored as **slot**.*Rules* and considers each rule in turn. For each rule it sets up the subtasks for each triplet of the if-slot, causing them to be executed in turn. If any of them fails, then the next rule in the list is considered. If the subtasks all succeed, then subtasks for each triplet of the then-slot are set up and executed. Nil is returned if all rules fail. The supplied frame name is used to bind variables where appropriate.

This then is an implementation of the basic expert system type of inferencing, where all actions are expressed in terms of frames, slots and values.

5.6 Other Means for Determining Values in FrameUP

The use of rules is not the only means available for computing values of slots. In general, any function can be placed in the *ToCompute* slot of a slot-frame. The predefined function use-rules is just one of an assortment of possible ToCompute methods.

Two others which are defined in FrameUP provide interactive question answering services. The function suggest-to-user is used, for example, whenever the user is to be asked if a particular value is true for some **frame**.*slot.* Ask-from-user is used when the user is to be asked for a value for some **frame**.*slot.*

The ability to specify any function at all for the value of the ToCompute slot allows great flexibility. One can specify that the relevant rules be tried first followed by asking the user. If both fail then perhaps an algorithm might be called upon. Or perhaps what is required is a value from some external data base. The usefulness of this kind of flexibility has been demonstrated in the GEM expert system (Williams, Nanninga and Davis, 1986).

6. AN EXAMPLE — HEFFE

A typical goal of a consultation with a diagnostic type expert system is to determine why some object is not operating. The goal in the following example is to determine why the object DeskLamp1 (of Figure 1) is not

operating. The example is drawn from the HEFFE system, an expert system programmed in FrameUP.

HEFFE begins by creating, under the direction of the user, a task of type FillSlot (Figure 5), which attempts to determine a value for the *ReasonForNotOperating* slot of **DeskLamp1**. This task will be placed on the agenda, and waits there to be selected for execution by the scheduler. Executing a task involves applying the *Procedure* for the particular type of task to the task in question. Figure 6 shows that this results in (get-value DeskLamp1 ReasonForNotOperating) being evaluated for this particular task.

There is no value stored for **DeskLamp1**.*ReasonForNotOperating* and so the function get-value calls upon the frame **ReasonForNotOperating** to supply the means for computing a value (Figure 9). This causes the rules to be used in an attempt to determine a value. It is also noted that the relevant rules are listed in this frame.

```
ReasonForNotOperating
    IsA: Slot
    Format: Singleton
    DataType: Atom
    Cache: Always
    MakesSenseFor: Object
    ToCompute: (lambda (fr)
                (use-rules fr 'ReasonForNotOperating))
    Rules: (Rule1 Rule3)
```

Figure 9. ReasonForNotOperating. Typically, a fault-finding system attempts to determine a value for this slot on some particular frame.

Rule1 may be chosen for execution. This rule (Figure 8) has two conditions, resulting in two tasks being created. These are Task2 and Task3 (Figures 10 and 11).

```
Task2
    IsA: Task
    TypeOfTask: CheckSlot
    ParentTask: Task1
    RelevantFrame: DeskLamp1
    RelevantSlot: Status
    RequiredSlotValue: Sound
```

Figure 10. Task 2 — Is the desk lamp sound?

```
Task3
    IsA: Task
    TypeOfTask: CheckSlot
    ParentTask: Task1
    RelevantFrame: DeskLamp1
    RelevantSlot: PluggedInto
    RequiredSlotValue: $PowerSupply
```

Figure 11. Task3 — Is the desk lamp connected to power point number 1.

Task2 is to check if sound is a value of **DeskLamp1**.*Status*. Finding that it isn't (Figure 1), the function stored on **Status**.*ToCompute* (Figure 12) is applied to the atom DeskLamp1. Again the rules are to be tried, with Rule2 (Figure 13) being chosen from the three candidate rules. The premise of this rule results in another task being created (Figure 14), which is to determine if DeskLamp1 operates from some PowerSupply. Suppose, for simplicity, that **Operates From**.*To Compute* specifies that the user be asked for a value, and the user supplies the atom PowerPoint2. Task5 is then created to handle the conclusion part of Rule2 which deduces that DeskLamp1 is Sound. Evaluating Task5 results in Sound being stored on **DeskLamp1**.*Status*. Thus Task2 has successfully been completed.

```
Status
    IsA: Slot
    Description: "Current known status of object"
    Format: List
    DataType: Atom
    Cache: Always
    ToCompute: (lambda (fr)
                (or (use-rules fr 'Status)
                    (ask-from-user fr 'Status)))
    Rules: (Rule2 Rule4 Rule1)
```

Figure 12. Status — A slot-frme.

```
Rule2
    IsA: Rule
    If: (($Object OperatesFrom $PowerSupply))
    Then: (($Object Status Sound))
```

Figure 13. Rule 2 — Determine whether an object is sound.

```
Task4
    IsA: Task
    TypeOfTask: CheckSlot
    ParentTask: Task2
    RelevantFrame: DeskLamp1
    RelevantSlot: OperatesFrom
    RequiredSlotValue: $PowerSupply
```

Figure 14. Task4 — does the desk lamp work when plugged into another power point.

```
Task5
    IsA: Task
    TypeOfTask: FillSlotWithGivenValue
    ParentTask: Task2
    RelevantFrame: DeskLamp1
    RelevantSlot: Status
    GivenSlotValue: Sound
```

Figure 15. Task5 — The desk lamp is sound.

Task 3 is to determine the value of **DeskLamp1**.*PluggedInto*. The value found there is PowerPoint1, which is bound to the variable $PowerSupply. The task succeeds.

Task2 and Task3 having succeeded, two new tasks, corresponding to the conclusion part of Rule1 are created. Task6 corresponds to having deduced that the reason for DeskLamp1 not operating is that PowerPoint1 is, for some reason, not operating. Task6 will return the value PowerPoint1. Similarly, Task7 corresponds to having deduced that PowerPoint1 is not operating.

```
Task6
    IsA: Task
    TypeOfTask: FillSlotWithGivenValue
    ParentTask: Task1
    RelevantFrame: DeskLamp1
    RelevantSlot: ReasonForNotOperating
    GivenSlotValue: PowerPoint1
```

Figure 16. Task6 — Return a value to the calling rule.

```
Task7
    IsA: Task
    TypeOfTask: FillSlotWithGivenValue
    ParentTask: Task1
    RelevantFrame: PowerPoint1
    RelevantSlot: Status
    GivenSlotValue: NotOperating
```

Figure 17. Task7 — Store a given value in the given slot.

All tasks associated with Rule1 have now successfully completed, and so Task1 has succeeded in determining PowerPoint1 as the value of *ReasonForNotOperating* for **DeskLamp1.**

The structure of the above sample execution of Task1 is summarised in Figure 18. To sum up, we have seen that to determine why DeskLamp1 is not operating we use Rule1 to deduce that it must be because PowerPoint1 is not operating. This is due to the fact that DeskLamp1 can be shown to be sound, using Rule2. This involved asking the user whether the desk lamp operates from some other power point.

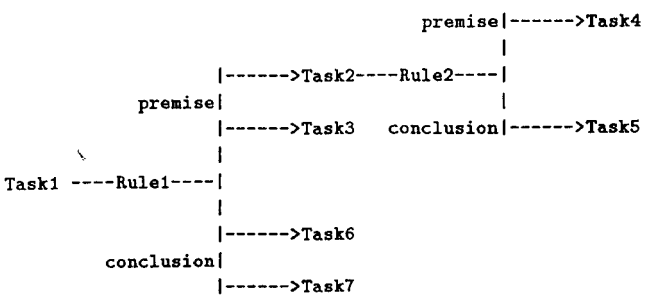

Figure 18. Summary of the tree of tasks.

The next task for the expert system will be to determine why PowerPoint1 is not operating. A similar approach will be taken in executing this task, using any appropriate rules.

7. SUMMARY

FrameUP is a frames-based system for the representation of knowledge in expert systems. It provides the machinery for representing, amongst other things, objects, rules and task structures. It provides, through the task and ToCompute structures, a flexible and easily modifiable mechanism for the solving of problems which are posed in terms of frames and slots. Various types of knowledge are also made quite explicit, through, for example, associating rules with particular frames, and with particular slots on the frames. The resulting flexibility is an essential facility for expert systems.

The hierarchical structure of frame-based systems has provided a natural structure for the storage of knowledge, allowing for the inheritance of rules, slot values and the ToCompute mechanism. The hierarchical structure also provides contextual information into which knowledge can be appropriately slotted.

The use of slot frames-provides a mechanism for specifying the fine control over the operation of the system. It is possible, for example, to independently specify how inheritance is to be used, and how to retrieve, store and compute values for individual slots. Information is neatly packaged providing easy access for modification to the system. Also, a variety of problem solving methods can be employed by the system.

REFERENCES

BARR, A. and FEIGENBAUM, E.A. (Eds.) (1982): *The handbook of artificial intelligence (Volume 2).* Los Altos, California: William Kaufmann.

BARSTOW, D.R. *et al.* (1983): Languages and tools for knowledge engineering. In F. Hayes-Roth, D.A.Waterman and D.B. Lenat (Eds.), *Building Expert Systems.* Reading, Massachusetts: Addison-Wesley.

DAVIS, R. and KING, J.J. (1984): The origin of rule-based systems in AI. In B.G. Buchanan and E.H. Shortliffe (Eds.), *Rule-based expert systems.* Reading, MA: Addison-Wesley.

GREINER, R. and LENAT, D.B. (1980): A representation language language. In *Proceedings of the First National Conference on Artificial Intelligence,* Stanford, California: AAAI-80.

HAYES-ROTH, F., WATERMAN, D.A. and LENAT, D.B. (1983): An overview of expert systems. In F. Hayes-Roth, D.A. Waterman and D.B. Lenat (Eds.), *Building expert systems.* Reading, MA: Addison-Wesley.

LENAT, D.B., HAYES-ROTH, F. and KLAHR, P. (1983): Cognitive economy in a fluid task environment. In R.S. Michalski (Ed.), *Proceedings of the International Machine Learning Workshop.* Monticello, Illinois.

MINSKY, M.A. (1975): A framework for representing knowledge. In P.H. Winston (Ed.), *The psychology of computer vision.* New York: McGraw-Hill.

WILLIAMS, G.J. (1986): Deficiencies of expert systems and attempts at improvement: A survey. In *Proceedings of the Ninth Australian Computer Science Conference,* Canberra: ACSC9.

WILLIAMS, G.J., NANNINGA, P.M. and DAVIS, J.R. (1986): GEM: A micro-computer based expert system for geographic domains. In *Proceedings of the Sixth International Workshop on Expert Systems and Their Applications.* Avignon, France.

WATERMAN, D.A. (1986): A Guide to Expert Systems, Reading, MA: Addison-Wesley.

BIOGRAPHICAL NOTE

Graham Williams is currently a senior tutor with the Department of Computer Science, Australian National University, where he is also a doctoral candidate. His research focusses upon the acquisition and representation of knowledge for use in expert systems. He has been involved in the development of a number of commercial and research expert systems. Mr Williams is also a consultant to HiSoft Computers' Expert Systems Division.