UFL: An Experimental Frame Language Based on Abstract Data Types

S. J. YOUNG AND C. PROCTOR
Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ

This paper describes an experimental language called UFL which has been designed to facilitate the programming of frame-based systems. In UFL, frames are treated as abstract data types. Each frame has a standard interface in the form of create, instantiate, read and write procedures allowing it to be given its own unique characteristics independent of the environment in which it will be used. Frames in UFL thus contain self-knowledge about how to create, instantiate and access themselves as well as the more usual application-level knowledge. This leads naturally to a language which is both extensible and powerful. Furthermore, UFL is entirely data-driven, making it particularly suitable for the implementation of intelligent knowledge-based systems.

Received June 1984

1. INTRODUCTION

The use of frames for representing knowledge in Artificial Intelligence applications is now a well-established technique. Essentially, a frame is a packet of information relevant to one particular concept or entity in a knowledge base. Frames are linked together by an inheritance hierarchy which enables frames representing ‘specialisations’ of a concept to inherit features from higher-level frames representing ‘generalisations’ of that concept. The contents of a frame may be simple atomic values, procedures (often called demons or methods in the literature) or subframes. The frame formalism provides a basis for representing knowledge in an efficient natural way and appears to be applicable to a variety of real-world problems.

There are several existing programming languages which provide direct support for the frame formalism such as KRL, FRL, UNITS and KL-ONE. However, these languages were all produced as research tools for investigating issues in knowledge representation and each has its own individual approach. Furthermore, in these languages frames are seen primarily as a method of organising the central knowledge data structure. Although they may have active components in the form of attached procedures, they are nevertheless primarily passive data structures operated on by external processing agents. For example, the FRL language is really just a library of LISP functions for performing basic frame operations on lists of objects organised in a frame-like manner.

This paper describes an experimental language called UFL which has been designed to support the programming of frame-based systems. It is a small, portable language which can be easily installed on any machine equipped with a standard Pascal compiler. Within the context of the frame formalism, UFL is a general-purpose language since it makes no commitment to any particular representational style. Furthermore, UFL is novel in that it is data-driven and self-defining. In UFL, frames are treated as abstract data types. Each frame 'knows' how to create and instantiate itself and does so when required automatically. There is no external agent in UFL, the system of frames defined by the programmer constitutes the entire program specifying both the structure of the data and its execution time behaviour.

2. BASIC PRINCIPLES

A UFL program consists of one or more frame definitions. Each frame definition denotes a named set of data and its associated operations. A frame in UFL is therefore analogous to an abstract data type in a conventional language. For example, consider the following package specification in the programming language Ada.

```plaintext
package PERSON_TYPE is
  type PERSON is private;
  procedure INIT(P : in out PERSON);
private
  type PERSON is
    record
      NAME : PERSON_NAME;
      AGE : INTEGER;
    end;
end;
```

This package defines an abstract data type PERSON with subcomponents NAME and AGE and an operation INIT to initialise objects of that type. In practice, of course, more operations would be required, nevertheless it is sufficient for the purposes of this example. Once the package has been defined, objects of type PERSON can be declared and initialised by calling the INIT procedure as, for example, in the following program outline.

```plaintext
procedure MAIN_PROGRAM is
  P : PERSON;
begin
  INIT(P);
  -- apply operations to it
end;
```

Notice from this that the instantiation of data objects and execution of the statement body are kept quite separate. This is typical of procedural programs, where the execution behaviour is governed entirely by the control flow programmed into it. The instantiation of data objects is of minor concern and the programmer, in general, has little control over it. In contrast, UFL provides a similar facility for representing data types but it is a data-driven language. There is no equivalent to the statement body in UFL and the execution behaviour of a UFL program is governed solely by the instantiation
of data objects; any procedural behaviour is subservient to this process.

To move from the procedural paradigm of languages such as Ada to the data-driven paradigm of UFL therefore requires a change in perspective. To make this change, consider again the example above. Suppose that the INIT procedure was called automatically following the creation of the data object P and furthermore, suppose that any other processing activities associated with P were handled by the INIT procedure. If this were the case, then there would be no reason to have a statement body to the program at all, since all processing activities associated with P would be embedded within its instantiation.

This changed view of program behaviour provides the starting point for describing UFL. The following UFL frame definition is a rough equivalent to the PERSON type given earlier.

```
person
(* name : person_name,
  * age : int,
  create std_create,
  inst std_inst,
  read std_read,
  write std_write )
```

A frame in UFL consists of one or more slots and is specified by writing a list of slot definitions separated by commas and enclosed in parentheses. Each slot definition specifies the name of the slot and its value. Thus the person frame contains six slots called name, age, create, inst, read and write. There are several different kinds of slot value possible within a frame. In the above, person_name and int are assumed to be the names of other frames, hence the values of the name and age slots are themselves frames. In these two slot definitions, the colon indicates that the value of the slot will be a copy or instance of the specified frame type. For example, the value of name will be an instance of the person_name frame. Note here that although it is convenient to refer to frame types and frame values, in practice UFL distinguishes between only through context. An instantiated frame value is structurally indistinguishable from a frame type and may itself be used as a frame type in a subsequent instantiation.

The slots create, inst, read and write have values which are procedures. These slots have a special status since they are the only slots which every frame must have (actually there are a few more slots in this category but they are not relevant to the discussion here). The procedures which fill these slots are called 'interface procedures'. They define the interface between the frame and its environment and control the way that any instance of the frame is created, instantiated and subsequently accessed. It is essential that every frame has these procedures because there is no external controlling agent in UFL. For example, suppose that an instance of the person frame is required, such as in the slot definition

```
client: person
```

In order to instantiate this frame, it must first be created. This is done by calling the create procedure of the client frame. This procedure will decide that the name and age slots must be created, so it will call the create procedures of those frames. The creation of these subframes is, however, the responsibility of the frames themselves. The create of client does not know or care what the create of name and age actually do, it only insists that they exist.

Basic frame instantiation is actually controlled by the inst procedure. In general, the instantiation of a frame simply involves the instantiation of each of its needed slots. These are slots marked by stars (asterisks). They denote the characteristic values of a frame, that is to say, those slots that must normally be instantiated because they characterise or distinguish one instance of the frame from another. In the example, the needed slots of person are name and age. Hence the inst procedures of these two subframes will be called by the inst of person. If these subframes themselves contained needed subframes, then their inst slots would be executed and so on, recursively. The recursion is terminated when an atomic or procedure slot is reached. In the former case, an actual atomic value must be obtained and written into the slot (e.g. by asking the user to type in the value from the terminal), and in the latter case the procedure is simply executed.

When a frame is written to (e.g. to assign a value to one of its slots), the values to be written are passed to the procedure in its write slot. Similarly, when a frame is to be read (e.g. in order to print its value), the procedure in the read slot is executed.

The reason for making frame-system execution totally dependent on these interfaces procedures is that it gives UFL great flexibility. In a sense, UFL is self-defining. The UFL programmer can write his own interface procedures and then define frames using these procedures.

Clearly as described so far, UFL would seem to be a cumbersome language to use, since it appears that a list of interface procedures must be given in every frame definition. This is not, however, the case. A major idea of the frame formalism is that of inheritance. In UFL, every frame must have an ako (a kind of slot) which defines its inheritance. For example, the predefined UFL environment includes a frame called standard which contains all the interface procedures needed to implement standard frame behaviour. Hence the person frame given earlier would actually be defined as follows.

```
person
(ako standard,
  * name : person_name,
  * age : int )
```

When an attempt is made to instantiate this new person frame, the system searches for an inst slot in person, and when this fails it searches back through the inheritance of the frame using the ako slot and finds the required slot in standard. The inheritance hierarchy of a frame can extend through any number of levels. This not only allows a convenient access path to the required interface procedures but also allows 'specialisations' of a frame to be defined simply and concisely.

To summarise, UFL treats frame definitions as abstract data types. Every frame contains a set of starred slots denoting the characteristic values of the frame and a set of unstarred slots representing the necessary background information needed to instantiate and use the frame. These slots are either given explicitly within the frame or inherited from higher-level frames via an ako slot. The predefined UFL environment contains a set of basic system frames such as standard for describing the
most commonly needed frame behaviours. This predefined environment is called the ‘base language’ and is itself written (mainly) in UFL.

Having described the basic principles of UFL, the remainder of this paper explains the language in more detail.

3. FRAME DEFINITION

As explained in the previous section, a frame definition specifies a record-like data structure consisting of a number of slots. Each slot has an optional name and a value. When a name is omitted from a slot definition then that slot is said to be ‘unnamed’. Unnamed slots are used in a variety of contexts where a positional rather than a named association of a slot value is needed. This is explained further later. Slot values are either atomic values, structured values in the form of subframes, references to other slots or procedures to execute.

Atomic values are either denotations or integers. A denotation is any name beginning with an upper-case letter or enclosed in quotes, and integers are written in the usual way. To support the use of atomic values, the UFL base language provides three atomic frame types called int, enum and bool. Each of these contains a ‘value’ slot for the actual atomic value plus slots defining the usual arithmetic, relational and logical operations needed to manipulate them.

There are two basic mechanisms used in defining a frame. First, a frame can be defined by explicitly enumerating its slots, and secondly, it can be defined as being an instance of an existing frame. These will be referred to as ‘explicit definitions’ and ‘instance definitions’, respectively. As an example, the person frame given in section 2 demonstrated both mechanisms

```plaintext
person (ako standard, 
  * name : person_name, 
  * age : int
)
```

Here the value of person is a frame defined by explicit enumeration of its slots; it is an explicit definition. Within the person frame, however, the slots name and age are also frames, but these are defined in terms of the instantiation (denoted by the ‘:’ symbol) of existing frames. These are instance definitions.

An instance definition can be augmented and modified by appending an explicit definition immediately after the frame type name. Hence, for example, in

```plaintext
rich_person : person ( * wealth : assets )
```

the frame rich_person is an instance of a person frame with an extra wealth slot added.

The augmented instance definition is a very powerful notational device which is widely used in UFL. It can be used to add further slots to instantiated frames as in the above example, or it can be used to fill in or replace existing slots. For example, consider the following time frame

```plaintext
time (ako standard, 
  * hour : init(0..23), 
  * min : init(0..59)
)
```

(continued from the previous page)

Here the value slot of hour is filled in with 12, and the same slot in min is filled in with 0 (the keyword const here simply makes the slot noon read-only). However, there are two problems with this definition. First, it is cumbersome and secondly, it again contravenes the principle of abstraction since the user of the time frame must have detailed knowledge of its internal structure in order to write values into it.

These problems can be avoided in UFL by using unnamed slots. The usual definition of the noon slot would, in fact, be

```plaintext
noon const : time(12,0)
```

In this case, UFL assumes that each unnamed value is to be written into the needed slots of the frame. Thus 12 will be written into the needed slot of the hour frame and 0 will be written into the needed slot of the min frame.

This mechanism is implemented by the write procedures of each frame, which like all UFL interface procedures operate recursively. When a write procedure is passed a frame value to write into its own frame, it takes each unnamed slot of that value and passes each in turn to the write procedures of each of its needed slots. Since this is recursive, it allows a list of values to be passed to a frame independently of that frame’s internal structure. Thus in the above case the values 12 and 0 would first be passed to the write procedure of the noon frame, which would then pass each in turn to the write procedures of the hour and min frames as required.

4. PROCEDURES

As noted earlier, the value of a slot may be a procedure, and instantiation of such a slot simply causes the procedure to be executed. In fact, this is only strictly true of external procedures. These can be thought of as system calls, and UFL provides a set of them which currently includes all the basic interface procedures, basic input/output operations and various primitives for manipulating slots. Apart from these external procedures though, all procedures which are defined by the user are just frames, and the term procedure itself is just a convenient fiction used to describe a frame which is instantiated primarily for its effect rather than to build a data structure. Thus, in the following, the term ‘execute’ is used in place of the term ‘instantiate’ only to indicate that the slot referred to is being instantiated primarily for its procedural effect. Similarly, the term ‘procedure’ itself refers to a value which is either an external procedure or a frame whose instantiation is expected to cause some desired effect.

User-defined procedures are synthesised using a proc frame which is instantiated with a list of unnamed slots, and its action is to execute each slot in turn. In addition,
there is an if frame and a while frame, which emulate the conditional and iterative statements of conventional languages. Examples of these are given below.

Although procedure valued slots can occur in any context, in practice they are mainly used in association with the instantiation mechanism. When the inst procedure of a frame is called for the first time, it checks to see whether there is an if_needed slot, and if so the attached procedure is executed. This procedure can then be programmed to take any preparatory action necessary in order to instantiate the frame, such as asking the user to input a value or computing a value from existing data. When subsequently the instantiation of a frame has been completed, the inst procedure looks for a check slot, whose value should be a Boolean function. If the evaluation of this function yields true then an if_added slot is executed, otherwise an if_inconsistent slot is executed. If there is no check slot then the if_added slot is executed directly by default. If either of the if_added or if_inconsistent slots is omitted no action is performed. Hence these procedures can be included in a frame, when required, to perform consistency checks and any completion operations necessary for the frame.

As an example of the use of procedures, consider the following definition of a time frame:

```plaintext
time
  (ako standard,
   * hour : int
     (0 . . 23,
      check : required(hour < 17),
      if_inconsistent : proc
        (: output('The office closes at 5 pm'),
         `time.destroy'),
      if_added : proc
        (: if(hour < 9,
            then(hour ← hour + 12)
          )
         )
     )
   )
   * min : int(0 . . 59),
   if_needed ask
  )
```

This frame contains two needed slots hour and min, and is designed on the assumption that a valid time should be within office hours, i.e. between 9 a.m. and 5 p.m. Internally times are represented using the 24-hour clock.

When an attempt is made to instantiate this frame, the if_needed slot will be executed, and this contains an external procedure called ask. Execution of ask first prints the name of the current frame on the terminal and then waits for the user to type in a frame value. Suppose that the value (19, 30) is entered. In this case, the value 19 will be written into the hour slot and its check function executed. This is an instance of the required frame, whose default action is simply to evaluate the Boolean condition passed to it as parameter, in this case hour < 17. Note here that operator forms such as these are treated into normal frame format i.e. < (hour, 17) by the syntax analyzer and require no special treatment by the UFL interpreter. Evaluation of hour < 17 returns false, and hence the if_inconsistent slot is executed. This is a procedure which first outputs an error message and then calls a standard external procedure called destroy to destroy the partially instantiated time frame. In this call to destroy, the up-arrow denotes a slot reference and the notation time.destroy denotes the slot called destroy within the frame called time. The full significance of slot references is explained later; here it is only necessary to understand that when a slot is referenced in this way it causes it to be instantiated. Thus, if the slot is a procedure, as in this case, it is executed.

Having destroyed the partially instantiated time frame, the attempt to instantiate it then resumes from scratch with the execution of the if_needed procedure again. When the user supplies a legal value for hour, the if_added procedure is executed instead of if_inconsistent. This consists of a single if frame which, in this case, checks to see if the value of hour is less than 9, and if so assumes that the user is not using the 24-hour clock and so must be referring to an hour in the afternoon. Execution of the then slot of the if frame accordingly increments the supplied value of hour by 12. Note here that the `←' operator denotes assignment in UFL. Thus if (2,15) was written into the frame, it would be converted to (14,15).

An important feature to emphasize about the above is the uniform treatment of data and procedures within UFL. To illustrate this further, the following frame defines a Boolean function:

```plaintext
bool_function
  (ako standard
     result : bool,
     action : proc,
     read : proc('action, result.read)
   )
```

In this frame, the read slot has been replaced by a procedure which first executes an action slot and then executes the read procedure of the Boolean result slot. Once defined, it may be used to instantiate specific Boolean functions such as:

```plaintext
in_range : bool_function
  (• val : int,
   • lo : int,
   • hi : int,
   action : proc
     (result ← ((val > = lo) and (val < = hi))
   )
```

This function could then be used in any context requiring a Boolean value such as in:

```plaintext
: if : in_range(x, 0, 100)
  then . . .
  else . . .
```

When an if frame is instantiated, it reads the value of its unnamed slot in order to determine whether to instantiate (i.e. execute) the then slot or the else slot. In this case the unnamed slot is an instance of the in_range frame. However, before the slot can be read it must be instantiated. This uses the standard inst procedure which writes the unnamed slot values x, 0 and 100 into the needed slots val, lo and hi, respectively. Then the read procedure of in_range is executed, which does the range test and returns the appropriate Boolean value via the result slot.
5. SLOT REFERENCES AND NAME SCOPE RULES

In common with most conventional languages, the same identifier can be used in UFL to name several different slots. When a slot is referenced by name, the reference is evaluated from its point of occurrence by searching for a slot with the same name. The rules governing this search effectively define the scope of names within the language and are as follows.

(1) The current frame is searched.
(2) All frames in the inheritance hierarchy as defined by the ako slots are searched.
(3) Steps 1 and 2 are repeated starting from the parent (i.e. the enclosing frame) of the current frame. The first instance of the required slot that is encountered during this process terminates the search.

Steps 1 and 2 above imply that for the purpose of evaluating names, a frame is regarded as the concatenation of itself with all the frames inherited via its ako slot. Step 3 effectively gives UFL the block-structured name scope of most conventional languages, but note that there is no implied declaration before use constraint.

When the name of a slot appears in any context other than its own definition it denotes either an instance of the named slot or a reference (pointer) to it.

There are two ways to specify an instance of a slot. The first way is to write the name of the slot immediately after a colon as in

```
fred: person
```

In this case, the slot referred to by person is determined at compile time and this, therefore, corresponds to static binding. This notation, as has already been seen, is used primarily in frame definitions. The second way of specifying a slot instance is just to write the name of the slot without a preceding colon. For example, in

```
when hour
```

the value of the when slot is an instance or copy of the hour slot. Here, however, the slot referred to by the name hour is determined at run-time and so this form of slot instance involves dynamic binding.

A slot reference is denoted by a preceding up-arrow symbol as in, for example,

```
when hour
```

In this case, the slot referred to by the name hour is evaluated at run-time as previously. However, the value of the when slot is not now an instance of the hour slot but a pointer to it. As a result, the effect of a slot reference is to equivalence two slots. In the example, there is a single copy of same value which is shared by both the hour and the when slots.

The semantics of slot references and instances have two important consequences. First, any slot which is referenced at run-time must be instantiated immediately. Hence, referencing a frame-valued slot effectively forces a change of context in the sense that the flow of frame instantiation is modified. Similarly, referencing a procedure-valued slot causes the attached procedure to be executed. An example of this was given in the previous section with the bool function frame. Secondly, when a slot is instantiated whose value is an instance of a procedure, then the procedure is copied to that slot and then executed. Thus the procedure is executed in a new context different from that in which it was originally defined. If this is not required then a reference to the procedure can be used instead of creating an instance of it and the procedure is then executed in its original context.

6. PROGRAM STRUCTURE AND EXECUTION

As noted previously, a UFL program consists of a number of frame definitions. At the outermost level of the program, each of these frame definitions is preceded by the keyword FRAME. In order to initiate the instantiation process, one frame definition must be marked as needed, and the instantiation of this frame represents the top-level goal of the program.

As an example, the following is a simple UFL program which might serve as the basis of an intelligent interface between an automatic scheduling program for organising meetings and a user (see Ref. 10 for a practical example of this kind of application)

```
FRAME location : enum
    (Tearoom, Conference room, Office_L4, Office_L9),

FRAME person : enum
    (Steve, Caroline, Les),

FRAME time
    (ako standard,
    *hour : int(0.23),
    *min : int(0.59),
    noon const : time(12,0) ),

FRAME meeting
    (ako standard,
    *who : person,
    *where : location,
    *when : time ),

FRAME *request : meeting
    (if_added call_schdule)
```

The top-level goal of this program is to instantiate a frame called request which is an instance of a meeting frame. This frame is designed to acquire all of the information needed from the user in order to schedule a meeting. Once completed, the if_added slot is executed and calls a procedure call_scheduled, which is assumed to provide the necessary interface between the frame system and some external process such as a database management system.

The precise way in which a UFL program executes depends on a number of state indicators which are attached to each slot. Of these, there are three major states which affect frame execution. These are the needed, active and completed states. The needed state is set on all starred slots at compile time. When a slot is accessed during execution, the active state is set. When a slot is fully instantiated, the completed state is set.

Frame-system execution takes place in cycles where each cycle commences with the execution of the inst procedure of the top-level goal frame. The action of the inst procedure and all of the inst procedures held within nested subframes depends on the states of the slots within each procedure’s own frame. More specifically, an inst procedure performs one of the following actions each time that it is executed.
(1) If there are active slots in the frame, then the inst procedure of the most recently activated slot is called else...

(2) If there are no needed and completed slots in the frame and there is an if_needed procedure then it is executed else...

(3) If there are sufficient needed and completed slots in the frame then the check function is executed. If this returns True or there is no check function the frame as a whole is set completed, otherwise the if inconsistent procedure, if any, is executed. Note here that setting the completed state clears the active state.

(4) If none of the above applies the inst procedure of the next needed slot in the frame is called.

In (3), the definition of ‘sufficient’ depends on the particular inst procedure being used. For example, the standard inst requires that all needed slots be completed, whereas the option inst requires just one to be completed.

In addition to the above, the create procedure is called each time that a new, as yet uncreated, frame is first accessed. Thus create is called to make the top-level goal frame and then subsequently each time that a new subframe is first accessed. A point to note here is that a frame is never created until it needs to be. This is important for two reasons. First, it allows frame definitions to be recursive and secondly, it promotes efficient use of memory.

The effect of the above rules can be illustrated by examining the execution of the example program given above. In the first cycle, the inst procedure of request is called. This results in a nested call to the inst of who, since there is no active or needed and completed slot nor is there any if_needed slot. The inst of who, however, finds an if_needed slot by inheritance since the enum frame contains the slot

if_needed ‘ask

The default ASK procedure is a simple communication device for obtaining data from the user. Its execution results in the question

request. who?

being printed on the user terminal. The user can then reply by typing in a suitable frame value, e.g.

(Caroline)

This is passed to the write procedure of the who frame, which copies it into its atomic value slot. This leaves the request and request. who slots active and the request. who. value slot completed.

In the second cycle, the inst procedures of request and who are called following the chain of active slots. The inst of who finds its only needed slot completed so it looks for a check slot. There is none and neither is there an if_added slot, so it simply sets the who slot completed and terminates.

In the third cycle, the inst of request finds no active slots, but its who slot is needed and completed so it does not attempt to find an if_needed slot. Hence, it calls the inst procedure of its next needed slot where and the same process is repeated as above to find a value for the where slot. Subsequent cycles then proceed in a similar manner to find values for the when. hour and when. min slots.

The main point to note about the above execution strategy is that the system never commits itself to more than one atomic slot instantiation per cycle. This is essential, since such an instantiation may involve the activation of other slots and subsequent cycles must deal with their instantiation as a matter of highest priority. An interesting analogy here is to compare UFL execution with that of a concurrent programming language. Each frame in UFL may be regarded as a process which is executed when its active flag is set and processes (i.e. frames) are scheduled such that the most recently activated has the highest priority.

Although very simple, this short program illustrates some of the features which make UFL particularly useful for dialogue control, which is one of its main areas of application (see 10 and 11 for other examples of using frames in this area). Returning to the execution behaviour of the program, it should first be noted that the frame values entered by the user in response to prompts for information contained only unnamed slots. As mentioned in section 3, the write procedure of a frame handles a list of unnamed values by passing each value in the list to each of its needed slots in turn. Slots can, however, be named if required and in response to the question

request. who?

The response could equally well have been

(who(Caroline))

More interestingly though, a different slot entirely could have been written into. For example, if instead the reply had been

(when(hour(10))

the write procedure of who would have failed to find a when slot in its own frame and would therefore have passed the value back unchanged to its parent request. The write procedure of request would then write the value 10 into the when. hour slot as normal. Furthermore, since this operation has the side effect of setting the when slot active, the next slot to be instantiated would be the min slot and not the original who slot. This illustrates the way that the focus of the dialogue can be shifted by referencing other slots. (Note here that in practice these frame values would not be supplied directly by the user but would be the output of a natural language front-end.) Following this shift of focus, the system would only return to the original who slot when the when frame had been completed. This same mechanism also allows the user to give the information requested as well as further information which had not been explicitly requested. For example, in response to the question request. who, the frame value supplied could have been

(who(Caroline), when(hour(10))

In this case the who slot would be completed before the shift of focus to the when slot, so that once the when slot was completed the system would go on to the where topic immediately. These shifts of focus seem to correspond closely to the way that humans behave in natural dialogue and therefore constitute a major strength of UFL when used in this kind of application area.

If, instead of an actual value, only a slot name is provided in a write operation, it is treated as a slot
reference and that slot is read. Thus if, in response to a question for the time, the reply was

(\text{noon})

The value of the noon slot, that is

(hour(12), \text{min}(0))

would be read and then written into the \textit{when} slot. This is an example of the way that declarative background knowledge can be built into a frame to aid in the interpretation of user inputs. The \textit{time} frame given as an example in section 4 also showed how procedural background knowledge can be programmed into frames. Hence UFL supports both declarative and procedural forms of knowledge representation.

Space does not permit a fuller discussion of UFL’s use in this area, but it is hoped that the above gives an indication of its potential.

7. IMPLEMENTATION

A UFL program is executed in two stages. First a compiler converts each frame definition in the source program into an internal representation, and secondly an interpreter is invoked to execute the compiled frames. The output of the compiler can be passed directly to the interpreter or used to provide an environment for further source compilations. Usually, a pre-compiled base language provides the initial environment. Both the compiler and the interpreter are written in Pascal.

Internally, frames are represented as linked lists of slots. The first slot of every frame is always the \textit{ako} slot and contains a pointer to the \textit{a_kind_of} frame. Each slot is a record with three fields containing the name, states and the value of the slot. The name is simply an index into a name table. The states include the \textit{active}, \textit{needed} and \textit{completed} flags referred to in section 6, as well as further states needed by the system. In particular, there is an \textit{unmade} flag which indicates when a frame needs to be created, and a group of flags which identify the kind of value in the value field. The value of a slot is either a pointer to a frame, a path reference or an atom. Path references result from slot references occurring in the source program which cannot be evaluated at compile time. They are represented by linked lists of slot names. An atom is either a simple integer, a denotation (index into the name table) or an external call (index into the procedure table of the interpreter).

The interpreter consists essentially of a built-in path de-referencing algorithm and a set of basic frame-handling procedures which are invoked at run-time by the instantiation of slots containing external system calls. All the real work is done by the interface procedures and their supporting system calls. The basic way in which these operate should be clear from the descriptions given in the preceding sections.

8. CONCLUSIONS

This paper has described the main features of an experimental language called UFL, which has been developed to support the programming of frame-based systems. This last section offers some final comments on its design and use.

As has been shown, UFL in common with other frame languages provides the two key mechanisms which characterise the frame formalism: inheritance and procedural attachment. In addition, UFL embodies two unique features. First, frame instantiation is controlled by the frames themselves and not by external agents. This leads to a data-driven style of programming which appears to be particularly effective for intelligent knowledge-based systems. Secondly, UFL is self-defining and extensible. There are few, if any, other languages which allow the fundamental operations of creation, instantiation and access to be redefined by the programmer. It is this last feature which, more than anything else, gives UFL its uniqueness and power.

The experience gained so far in using UFL has been limited to the construction of the language comprehension and dialogue control components of a speech-understanding system. This experience has been encouraging, but it has also identified two problems with the language. First, the dynamic name scope rules for instances and references are too liberal. When a slot of a frame is referred to from within a nested frame then the system searches the \textit{ako} hierarchy of that nested frame. This means that if the name was used within the hierarchy the wrong slot is found. The effect of this is that the applications programmer often has to avoid using names which are also used in higher-level frames and, in particular, in the base language. This is clearly undesirable since it violates the principle of abstraction on which the language is based. The problem could be avoided by using special naming conventions in the base language, but this would only be a partial solution and since the current name scope rules are also difficult to implement efficiently, work is now being done to provide clearer and more efficient name-binding mechanisms.

The second problem concerns the definition of procedures, and particularly interface procedures, within the UFL language. As has already been emphasised, the ability to selectively redefine interface procedures such as \textit{inst}, \textit{read} and \textit{write} is a major strength of UFL. An example of this was given in section 4, where a function frame was created simply by redefining the \textit{read} slot of a standard frame. This example was, however, not typical in the sense that it did not require any searching of the frame structure. In contrast, more general cases such as the definition of \textit{inst} procedures depend almost entirely on searching for slots with particular names and states, and these operations have proved to be both cumbersome and inefficient to implement using conventional \textit{proc}, \textit{if} and \textit{while} constructs. As a result, work is now proceeding to build a search and pattern-matching mechanism directly into UFL designed specifically for searching and manipulating frames.

Acknowledgement

Acknowledgement is made to British Telecom for funding this work and to the Director of Research of British Telecom for permission to publish this paper.
REFERENCES


Announcement

20–24 OCTOBER 1978

The Third International Conference on Text Processing Systems PROTEXT III Conference (22–24 October 1986) and the related PROTEXT III Short Course (Computer Aided Text Processing – An Introduction, 20–21 October 1986) will be held in the University Industry Centre, University College, Belfield, Dublin and are sponsored by the Institute for Numerical Computation and Analysis, Dublin.

PROTEXT III Conference Keynote Speakers:
C. BIGELOW (Bigelow and Holmes, San Francisco): ‘Typeface transformations and deformations’.
P. MacKAY (University of Washington, Seattle): ‘Experience in design with Metafont’.
R. MORRIS (Interleaf University of Massachusetts, Boston): ‘Evolution of the Interleaf user interface’.
A. SHAW (University of Washington, Seattle): ‘Structure editor generators for documents, programs and structured data’.

PROTEXT III Short Course Lectures:
K. HOLMES, ‘Fundamentals of digital alphabet design and editing’ (2 lectures).
P. MCKAY, ‘Introduction to TeX’ and ‘Introduction to LaTeX’.
R. M. MORRIS, ‘Introduction to PostScript’ (2 lectures).
S. SMITH, ‘Using the SGML starter set’ and ‘Expanding the SGML starter document type definition’.

Registration Fees
For each event, the applicable fee covers lunch and refreshments daily, relevant documentation and entry to all sessions. In addition the short course fee includes the Welcome Reception on 19 October and a copy of the Lecture Notes (published before the short course), while the conference fee includes the Conference Dinner on 22 October and a copy of the Proceedings (published after the conference).

There is no charge for Associates of delegates and they are welcome to attend all Social Events but they may not attend any of the Technical Sessions or appear as joint authors on papers. Full-time bona fide research students are encouraged to contribute papers to the Conference and they may make a written application in advance for a scholarship to cover part of their registration fees. Each case will be considered on its merits; normally a faculty member from the student’s institution should be a registered delegate. The early rate applies to all fees received by 1 June 1986. Please note that refunds can be made only for cancellations received by us, in writing, before 1 August 1986.

Short course or conference (early), US$ 250 (late) US$ 295; short course and conference (early), US$ 350 (late), US$ 395.

Registration

Registration will take place as indicated below on Sunday, 19 October in the Montrose Hotel and on subsequent days in the University Industry Centre, University College.

Sunday, 19 October, 15.00–20.00; Monday, 20 October, 08.00–17.30; Tuesday, 21 October, 09.00–19.00; Wednesday, 22 October, 08.00–17.30; Thursday, 23 October, 09.00–17.30; Friday, 24 October, 09.00–16.30.

Social Events
19 October: Evening Welcome Reception in the Montrose Hotel.
22 October: Conference dinner, including wine, in the Hibernian United Service Club. Tickets for Associates US$25 per person.
Advance booking for above is required and refunds can be made only for cancellations received by us, in writing, before 1 September 1986.

Hotel accommodation

The Conference organisers have reserved ample accommodation, however the availability of rooms cannot be guaranteed after 1 September. We will of course, try to accommodate late bookings. Please note that under no circumstances can accommodation be reserved without a deposit of US$75 per person; also that refunds can be made only for cancellations received by us, in writing, before 1 September 1986. Rates quoted are per person per night and include full breakfast, service charges and Government taxes applicable at time of going to print; should any increase or currency fluctuation occur our rates will be amended accordingly. The Montrose Hotel is a Grade A hotel 10 minutes walk from the conference venue. All bedrooms have private bathrooms, telephone, television and radio. Single: US$49, Sharing Twin: US$35. For those delegates requiring more economical accommodation the conference organisers will arrange to accommodate them in nearby guest houses. These cost approximately US$20 per person per night, including full breakfast, plus a booking fee of US$5 per booking. Please note that usually guest house accommodation will be without private bath or shower and will not have any restaurant facilities. The exact location and applicable cost will be advised when guest house accommodation is confirmed to individual delegates.

Late papers

To enable participants to report on their latest research a session for late papers will be organised. These can be accepted up to the last minute but there is no guarantee that they will be published in the Conference Proceedings.

Exhibitions and photocopying service

A table will be provided for the display of relevant preprints and offprints. You are encouraged to bring your own and your colleagues’ material for display. A photocopying machine will be available nearby so that others may copy it. An exhibition of books, journals and promotional material for other relevant conferences will also be held.

For further information contact:
PROTEXT III, Conference Management Services, P.O. Box 5, 51 Sandyvore Road, Dún Laoghaire, Co. Dublin, Ireland.
Tel: (+353) 1-8090205 Telex: 30547 SHCN EI (Ref. BOOLE)

Telegrams: BOREPRESS DUBLIN EIRMAIL (Dialogon) 74: EIM194
To contact Professor Miller you may use:
EUROKOM: JOHN MILLER TCD INTERNET: JMLMILLER@UCL-CS

THE COMPUTER JOURNAL, VOL. 29, NO. 4, 1986 347