The Operational Feature Exchange Language

David Alan Bourne, Jeff Baird,
Paul Erion, and Duane T. Williams

CMU-RI-TR-90-06

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

March 1990

Copyright © 1990 Carnegie Mellon University

The work described in these documents was supported by Air Force Contract No. F-33615-86-C-5038, Intelligent Machining Workstation (IMW), sponsored by the Air Force Wright Aeronautical Laboratories, Materials Laboratory, Air Force Systems Command, Wright-Patterson Air Force Base, Ohio.

Contents

1 Executive Summary 1

2 FEL Specification: Preliminary Design 5

3 FEL Interface to the Planning Expert 31

4 FEL Syntax for Communicating with a Geometric Modeler 53

5 FEL Interface for Communicating with the Holding Expert 75

6 Generic Environment for Unix-based Experts 97

7 Generic Environment for Lisp-based Experts 113

Abstract

This collection of documents describes the existing Feature Exchange Language (FEL) as it is implemented in the prototype Intelligent Machining Workstation (IMW). The original design of FEL is described, the FEL interface to several IMW subsystems (Planning, Modeling, and Holding) is explained in considerable detail, and the implementations for both the Sun Unix C++ and the TI Explorer Lisp environments are explained.

Executive Summary

The Feature Exchange Language (FEL) has been designed as a language for communicating messages in a distributed environment.

Design Feature-1: FEL was designed to make parsing and generation simple. The assumption was that possibly many different computing platforms would require an FEL interpreter.

Design Feature-2: It was assumed that FEL could tie together processes running on one machine or many, and therefore FEL was designed to represent the sender, receiver(s) and other information that could be useful in message transport.

Design Feature-3: Features were chosen as the objects that should be communicated. Each feature has a name and a set of built in parameters that go with it.

Design Feature-4: Each sentence of FEL has a verb that specifies how a message should be interpreted by a particular receiver. There has been a strong effort to keep the number of verbs to a minimum, allowing message variations to be specified in the feature-lists. Note: verbs are overloaded in the sense that each receiver can interpret a message in a way most meaningful to it.

Design Feature-5: FEL was designed to represent complex message exchanges between distributed parties. For example, verb-tense and a dialogue name are used to open, keep-track-of, and close a given dialogue. In this way, it is possible to essentially process multiple messages at once: every receiver can act as a server.

This document explains how these features were in fact implemented and then how FEL was used in various applications:

- 3D Modeler Updates and Queries
- Planning Requests for Information
- Holding Expert Queries and Answers

We believe that this language is a good first step towards the development of a messaging system that can easily integrate and coordinate software modules, especially those found in manufacturing applications.

There is further work that we believe is necessary to have a truly useful manufacturing language.

- Including feature definition as part of the language
- Extending work on dialogue management
- Supporting negotiation and contract management as part of an FEL implementation

Center for
Integrated Manufacturing Decision Systems

FEL Specification
Preliminary Design

David Alan Bourne

March, 1990

Abstract:

This document is the original (preliminary) design specification for FEL (Feature Exchange Language).

A detailed description of the original syntax of FEL sentences is given, as well as a detailed overview of the semantics of many of the verbs and attributes of the language. The current implementation of FEL was motivated by this document. Some of the advanced language features described here have yet to be implemented, but all of the basic features of the language are employed in the existing prototype of the IMW (Intelligent Machining Workstation).

Copyright © 1990 Carnegie Mellon University

Contact:

David Bourne
CIMDS
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-8810

1. Background Information

The Intelligent Machining Workstation includes many different subsystems, which must communicate with each other. Each subsystem has its own features, parameters and capabilities; so the Feature Exchange Language (FEL) specifies a simple, yet unified, approach for exchanging this information.

At the same time, FEL must cope with part descriptions or product definitions. The spirit of language family PDDI/GMAP/MFGMAP/PDES is to go beyond geometry and encompass process into the part description. Since MFGMAP is meant to be our subset of GMAP it is important to keep the same flavor to it – it must specify more than just geometry.

Requirements for both product definition and control information have two major parts: a command and many parameters. The parameters can be collected together to describe the objects that we wish to manipulate – these objects are called feature lists. Each of these feature lists describes a collection of attributes and values that constitute a feature definition; in some cases, it is more convenient to think of the feature lists as describing objects or nouns.

2. The Feature Exchange Language (FEL)

Since we have unified the goal of these two languages into one way of describing and exchanging features of parts and processes, we have decided to rename it the "Feature Exchange Language." The top-level syntax of this language is described as:

Sentence::=	"(" <Verb> <Feature_Lists> ")" | "(" ")"
Feature_Lists::=	<Feature_List> <Feature_Lists> | <Attrib> <Feature_Lists> | nil

The language syntax specifies that any command can have any number of feature lists. This allows verbs to be written such that they can take any number of objects (i.e., feature lists) as arguments. It is also possible to add additional feature lists that could describe where the message is sent, and notes that might be useful for debugging and user displays.

3. The FEL Verb Breakdown

In FEL there are five different types of each verb. These verb styles denote state transitions in a dialogue between two subsystems. The state transition tables for these dialogues are discussed in a later section.

(1) Present -- Command to go to new state
(2) Active -- In-process of going to new state

(3) Past -- Have reached new state
(4) Not_Past -- Have not and will not reach the desired state
(5) Stop_Active -- Terminate an active state

These verbs define transitions in a simple state network. The "imperative verbs" are equivalent to the "present verbs" and the "declarative" verbs are split into "active" and "past" verbs. The "interrogative" verbs are still covered by the "present verbs" by choosing verbs that request information.

There are also two control oriented verbs (Not_Past and Stop_Active). In any type of error situation it is necessary to communicate that an action has not been accomplished. On the other hand, it may be necessary to abort an ongoing action, because of some external influences or new information. This will be discussed further in the section on *communication states.*

The result is that each communication has a beginning, an optional middle and an end. For example, one interchange of verbs could be:

Message Out from Source:	-->	**Make**
Message Received at Source:	**Making**	<--
Message Received at Source:	**Made**	<--

The lexical analyzer can be responsible for marking the tokens as present, active or past. This scheme will work fine for most verbs, but may cause some confusion for irregular verbs, such as, "is", "being", and "was." There are two solutions to this problem. We can introduce a special characters, such as, "!", "<","=". In this case, "is!", "is<" and "is=". Or simply, we could choose built-in verbs with reasonable conjugations.

Verbs::= <Present> | <Active> | <Past> | <Not_Past> | <Stop_Active>

Recommendation: There are advantages to using English verbs and conjugations, because they can be readily paraphrased into full English sentences for the human interface.

A design goal for FEL is to make its verbs and verb-types orthogonal. That is, every verb should be able to be put into every form. This should be true no matter what verb classification scheme is chosen.

4. Built-In Verbs

In general, the verbs are meant to describe actions on feature lists as well as methods and states of negotiation. Fortunately, very few verbs will go a long way, because most of the final actions depend on the contents of the feature lists. What follows is a *nearly* complete list of verbs that will be needed for the IMW.

	Defined Example
Negotiation Verbs:	
Offer | Offering | Offered |Not_Offered | Stop_Offering	pp. 24-25
Accept | Accepting | Accepted | Not_Accepted	
Cancel | Canceling | Canceled | Not_Canceled	
Database Verbs:	
Define | Defining | Defined | Not_Defined	pp. 21-23
Update | Updating | Updating |Not_Updated	
Delete | Deleting | Deleted | Not_Deleted	p. 27
Get | Getting | Got |Not_Gotten	p. 26
Machining Verbs:	
Make | Making | Monitoring| Made | Monitored | Not_Made | Not_Monitored | Stop_Monitoring | Stop_Making	pp. 26-27
Inspect | Inspecting | Inspected | Not_Inspected | Stop_Inspecting	

5. *Verb Definitions*

Offer: This message verb is a command to fill in suggested values in a feature list. For example, if the planner wishes to ask the cutting expert for suggested speeds and feeds then the planner could use this verb.

Accept: This message verb agrees on negotiated values passed between two systems. It has the effect of signing a contract between the two systems after an offer has been made. Eventually, we may impose sanctions on systems that wish to break contracts.

Cancel: This message verb is the only way to break a contract that has been negotiated between two subsystems. If it is issued, then it is assumed that the system breaking the contract is subject to the sanctions of the contract.

Define: This message verb allows one subsystem to pass a feature list to another system, because it is known *a priori* that the information is needed. For example, this message type would be used to distribute part model information.

Update: This message verb allows one subsystem to update remote copies of a feature list. This can be used only if the remote copies of the feature list belong to the source of this message, otherwise there is a violation of data security.

Delete: This message verb allows one system to delete another system's feature list. This verb cannot be used to delete pending contracts, though once contracts have been completed, then this verb can be used.

Get: This message verb allows one system to get a feature list from another system. Ownership of the feature list remains with the source of the information.

Make: This message verb commands the system to carry out actions that will make the defined features in the external world. In other words, this verb would command the system to machine a part, rather than manipulating internal information.

Inspect: This message verb commands the system to carry out actions that would inspect the external world. In other words, after a part is machined, this verb could be used to inspect the final result.

6. Representing Parameters as Feature Lists

The primary problem area for both MFGMAP and SML was how to represent many, many complex parameters. SML chose a representation known as Working Elements based on work done at NBS. This scheme is essentially named structures, which can be nested into hierarchies. We represent these hierarchies by explicitly naming the nodes of the hierarchy and then list branches of the hierarchy as its parts with "Has_Parts."

Example Feature List:

```
(   (Name       PL_Cylinder)
    (Type       Cylinder)
    (Length     5 inches)
    (Radius     3 inches)
    (Has_Parts  (CX_Hole1 CX_Hole2  CX_Hole3)))
```

This example shows how the planner might describe a cylinder with three subparts as a feature list. Also nested feature lists do not need to define a proper hierarchy. For example, MFGMAP applications make it necessary to represent loops for edges and faces.

To make parsing as simple as possible, all special symbols have been removed and the syntax is essentially equivalent to association lists in LISP. However, the basic structure is identical to NBS-type work elements.

One problem with these feature lists is that all the different attributes have to be understood by each subsystem and yet they remain *ad hoc*. Therefore, as an area of development, we need to further systematize the attributes and consider computational devices for dynamically introducing new attributes.

7. Feature List Syntax

A partial syntax description of a feature list is:

Feature_List::=	"(" <Attrib_List> ")"
Attrib_List::=	"(" <Attrib> <Value> ")" <Attrib_List> | nil
Attrib::=	<Alphabetic+_> | <Alphabetic+_> <Alphanumeric+_>
Value::=	<Attrib> | <Integer> | <Floating Point> | <Dimensions> | | <Rates> | <String> | <List>
Dimensions::=	<Integer> <Units> | <Floating Point> <Units>
Units::=	"cm" | "mm" | "inches" | "mils" | "sec" | "min" "radians" | "degrees" | nil
Rates::=	<Integer> <Rate_Units> | <Floating_Point> <Rate_Units>
Rate_Units::=	"rpm" | "rps" | "ips" | "ipm" | nil
Value2::=	<Attrib> | <Integer> | <Floating Point> | <Dimensions> | | <Rates> | <String>
Val_List::=	<Value2> | <Value2> <Val_List>
List::=	"(" <Val_List> ")"

Syntax Notes: In this syntax, the "Name" attribute is required and internal lists are forced to be flat, that is, attribute values cannot be lists of lists. This later limitation is to avoid the temptation to hide feature lists inside other feature lists. This syntax assumes that the lexical analyzer uses any amount of *whitespace* as separators -- any number of spaces, tabs, line feeds or carriage returns. Some syntactic distinctions could be simplified, e.g., rates and dimensions, but these extra syntactic categories should simplify semantic interpretation.

There are many examples of this syntax in later sections of the document.

8. Built-In Feature List Types

Simple Machined Features	Defined
Block	√
Blind_Hole	
Thru_Hole	√
Hole_Chamfer	
Thread	
Slot	
Thru_Slot	√
Rectangular_Thru_Slot	
Rotational-Groove	
Pocket	
Thru_Pocket	
Rectangular_Pocket (?)	
Chamfer	
Channel	
Shoulder	
Squaring_Block (?)	
Plane	
Angle (?)	
Thru_Angle	
Convex_Angle	
Edge_Round	√
Wedge	√
Face	
Miscellaneous	
Part	√
Surface	√
Cutting_Operation	√
Holding_Operation	√
Sensing_Operation	√
Manager_Operation	√
Message	√

9. Feature List Definitions

Each built-in feature list has its own set of predefined attributes. Some of the predefined values must have a particular kind of value. Some of these value types are considered obvious, e.g., <Integer> and others are defined above as part of the feature list syntax. The definitions that have been predefined follow:

```
(   (Name          <Attrib>)
    (Type          Thru_Hole)
    (Radius        <Dimensions>)
    (Depth         <Dimensions>)
    (P_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (D_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (Has_Parts     <Value>)
    (Is_Part       <Value>))

(   (Name          <Attrib>)
    (Type          Thru_Slot)
    (Width         <Dimensions>)
    (Depth         <Dimensions>)
    (Length        <Dimensions>)
    (P_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (W_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (D_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (L_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (Has_Parts     (<Value>)
    (Is_Part       <Value>)))

(   (Name          <Attrib>)
    (Type          Wedge)
    (Length        <Dimensions>
    (X_Angle       <Dimensions>
    (Y_Angle       <Dimensions>
    (Intersection  <Dimensions>  ?? This this be a squared value
    (P_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (W_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (D_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (L_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (Has_Parts     <Value>)
    (Is_Part       <Value>)))
```

```
(   (Name          <Attrib>)
    (Type          Block)
    (Width         <Dimensions>)
    (Height        <Dimensions>)
    (Length        <Dimensions>)
    (P_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (W_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (D_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (L_Vector      (<Dimensions> <Dimensions> <Dimensions>))
    (Has_Parts     <Value>)
    (Is_Part       <Value>)))

(   (Name          <Attrib>)
    (Type          Edge_Round)
    (Radius        <Dimensions>)
    (Class         "Convex" | "Concave")
    (Has_Parts     <Value>)
    (Is_Part       <Value>))

(   (Name          <Attrib>)
    (Type          Part)
    (Material      <Materials>)
    (Units         <Units>
    (Source        <Batch_Codes>)
    (Surfaces      (<S_Type> <S_Type> <S_Type>  <S_Type> <S_Type>
                   <S_Type>)
    (Presentation  (<Side> <Side> <Side>))
    (Has_Parts     <Value>))

(   (Name          <Attrib>)
    (Type          Surface)
    (Class         "Sawed" |"Rolled" | "Machined")
    ??Tolerance
    (Has_Parts     <Value>)
    (Is_Part       <Value>))

(   (Name          <Attrib>)
    (Type          Cutting_Operation)
    (Units         <Rate_Units>)
    (NC_Name       <Attrib>)
    (NC_Frame      <Value>)   ; We might want to make this a separate element
    (Safe_Zone     <Value>)
    (Tools         <Value>)
    (Speed         <Rates>)
    (Feed          <Rates>) )
```

```
(   (Name          <Attrib>)
    (Type          Sensing_Operation)
    (Concerns      <Value>)
    (Warnings      <Value>)
    (Errors        <Value>))

(   (Name          <Attrib>)
    (Type          Holding_Operation)
    (Fixtures      <Value>)
    (Cutter_Paths  <Value>)
    (Forces        <Value>))

(   (Name          <Attrib>)
    (Type          Manager_Operation)
    (Models        <Value>)
    (Active_Element <Attrib>)
    (Concerns      <Value>))

(   (Name          <Attrib>)
    (Type          Message)
    (Reply_To      <Attrib>)
    (From          <System_Module>)
    (To            (<System_Modules>))
    (Priority      <0..255>)
    (Time          <Time_Stamp>)
    (Subject       <string>))
```

10. Special Attributes (Is_Part and Has_Parts)

There are two very special attributes called "Is_Part" and "Has_Parts." These two elements are used to tie the structure of feature lists together. The Is_Part relation is used to determine the structure of value inheritance, i.e., what are the default values. While the Has_Parts relation, is used to indicate that there are more details in other feature lists and it may be necessary to consider the whole. In the case of geometrical modeling, Has_Parts can be used to trigger modeling primitives that perform additions and subtractions from the model.

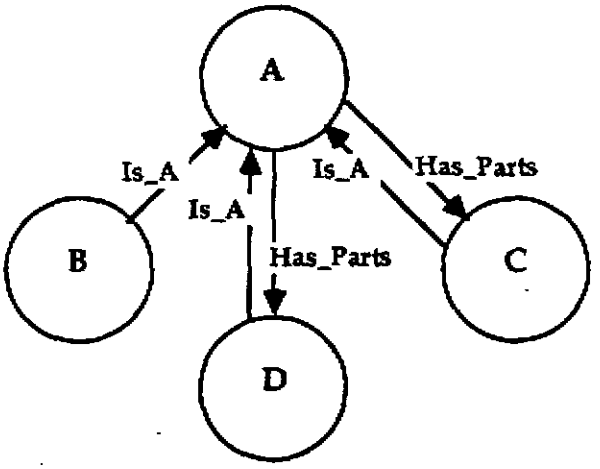


Figure 1: Four feature lists named: A,B,C and D

Figure 1 is a graphical illustration of the connection between four feature lists. In this case, we recognize that the Is_Part and Has_Parts relations are not symmetric, because "B" is not included as part of "A's" description. This hierarchy of feature lists could be defined as:

```
(Define   ((Name       A)
           (Has_Parts  (C D)))

          ((Name       B)
           (Is_Part    A))

          ((Name       C)
           (Is_Part    A))

          ((Name       D)
           (Is_Part    A)))
```

11. Parsing Feature Lists

On SUNS and other UNIX boxes, LEX and YACC can be used to define a parser for feature lists. The output of the parse will, in these cases, be stored in C structures.

On Explorers, the LISP reader will be used for reading the feature lists into LISP association lists.

12. Message Information

Both the *source* of a message and the *target* of the message should be represented. We can decide to include this in the FEL proper or we can let the mail system manage this information for us. Functionally, the result will be the same, but different modules will be responsible depending on the choice.

We may also want to add other features to messages, such as, carbon copies, forwarding and status required or not.

13. Feature List Naming Convention

The subsystem, name in italics, that constructs a feature list must mark it with a two letter prefix, which corresponds to the name of the module. In this way, it will be clear which module owns every feature list. This becomes important when we wish to maintain consistency between values in our distributed environment.

CT	--	CutTech Database
CX	--	Cutting Expert
HI	--	Human Interface
HX	--	Holding Expert
MX	--	Modeling Expert
PM	--	Plan Manager
PX	--	Planning Expert
SX	--	Sensing Expert
UP	--	Upstream Processes, e.g., Designer, Scheduler. We may want to break this category down further.

14. Convention for Protecting Distributed Data

The feature lists are meant to operate in a distributed environment, which can cause many database management problems. To avoid problems, like having two copies of the same feature list with different values, we will enforce a restriction that makes feature lists "read everywhere" and "write and delete only at the home subsystem." Even this restriction may be too weak, unless every subsystem remembers who has copies of that subsystem's feature lists. This means it is even necessary to control when the owner of the feature list is allowed to make changes.

For example, it is understood that a negotiation between subprocesses is a time for making changes. But once values have been jointly settled, then that feature list is effectively closed. To understand this better, a negotiation is like coming to terms on a contract between two parties. Once an agreement has been devised, then both parties are honor bound to that agreement and its terms. After the contract has been completed, then the feature list is obsolete and can be safely removed.

There is a further need to formalize when it is safe to modify existing feature lists, so that consistency can be guaranteed without giving up flexibility.

15. Dialogues and Communication States

The IMW is designed so that any two subsystems can carry out a dialogue at any time. These dialogues can have many purposes, which can vary from carrying out actual machining to simply settling on a plan. Whenever two systems participate in such a conversation, they must each maintain the current state of the dialogue. In fact, these states must be maintained for *every* active dialogue that is going on concurrently. Furthermore, it is expected that both participants in a dialogue maintain their own state network.

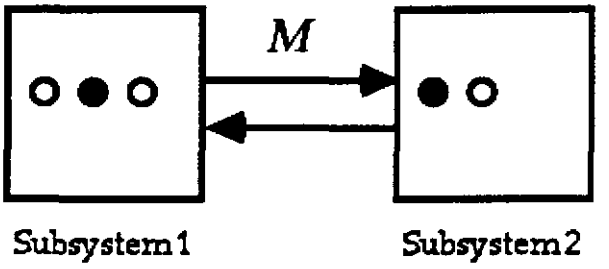


Figure 2: Two subsystems in dialogue

Figure 2 shows two subsystems with identical state machines, which represent the dialogue in process. The first subsystem which began in the first state, issued a message to the second subsystem and then advanced to the second state. The second subsystem remains in the first state until it receives the incoming message *M* after which it also advances to the second state. The dialogue between the two subsystems in not considered complete until both state machines return to the initial state. Practically speaking, the state machines are automatically created and purged as individual dialogues are started and then completed.

Most communications can be handled with a simple three state machine. Figure 3 shows a three state machine for one participant that starts in **S** and usually proceeds through two intermediate states in the process of completing a request. However, on occasion it is possible to determine that the request can be immediately satisfied or that it cannot be satisfied at all. In this case, the dialogue can be brought to an immediate end.

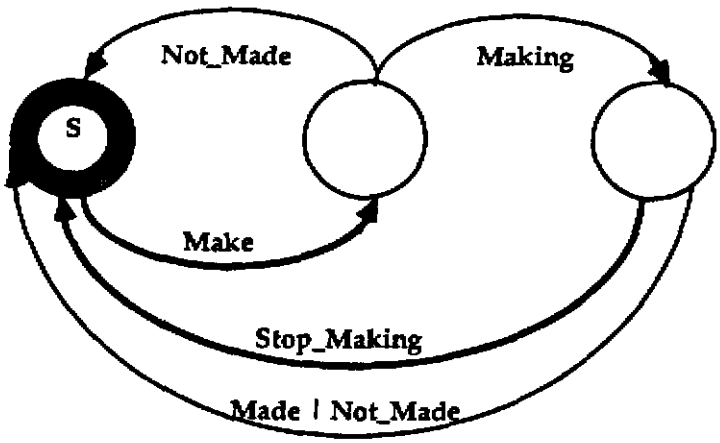


Figure 3: Simple Three state communications

In this network, there are two kinds of arcs: **bold** arcs indicate that the message is received and plain arcs represent that the message is sent to the other participant. Therefore, if a subsystem is the origin of a dialogue then it is allowed to initiate the **bold** arcs, while if a subsystem is the target of dialogue then that subsystem is allow to initiate the plain arcs. On the other hand, the origin of a dialogue is expecting to receive messages on plain arcs, while the target of a dialogue is expecting to receive messages on **bold** arcs. Figure 3 also raises a potential problem. Once a subsystem is in the active state, "making" in Figure 3, it may have to be forced out of it by the verb "Stop_Making," otherwise there would be no way to cancel an action once it had started.

These state machines are quite powerful, but cannot handle all of the complexities that we wish to capture in a negotiation process. However, Figure 4 shows that with the addition of one state it is possible to add a negotiation loop, between state 3 and state 4, and still be able to keep track of which subsystem has the proverbial ball.

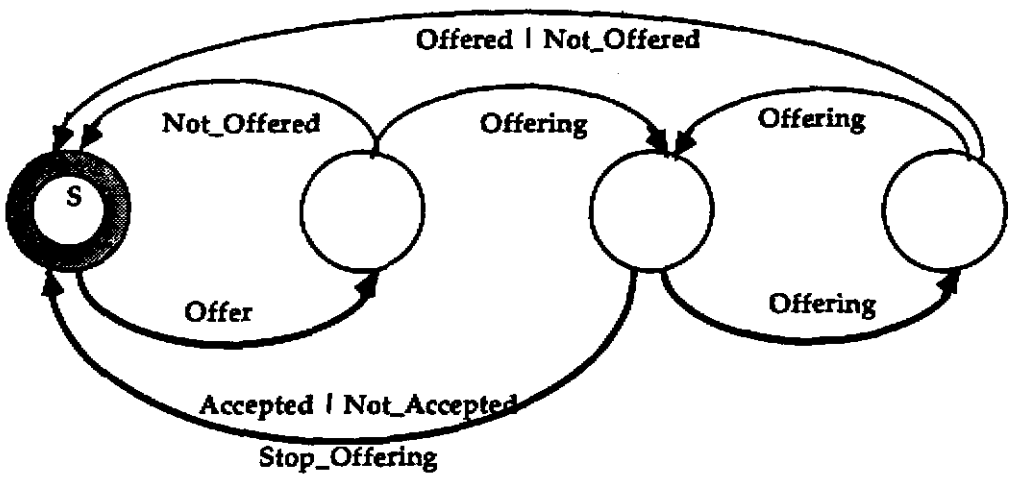


Figure 4: Complex Four state communications or negotiation

First, the origin of a negotiation decides that it requires information on several features. It then sends out a possibly empty feature list to be filled in by the source of the message. That is the origin requests an "offer" for possible values. When the source of the message calculates the appropriate values to return it is "offering" a response, one of possibly many. The source of the original request can then make a counter offer or simply state that the original offer has been "accepted."

Another strategy, see Figure 5, for a simple kind of negotiation can use another subsystem just to validate the terms of an agreement. In this case, an "accept" command is sent to a second system and a response comes back indicating whether or not the terms have been accepted.

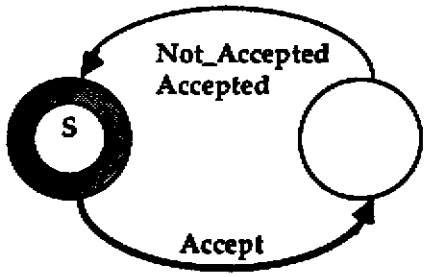


Figure 5: Simplest two state interaction

16. Modeling Stock Example

This long example starts with one subsystem called a **planner** sending part information onto other systems that will also need the information. Since this information was not requested, the "define" verb is used. The idea is that each subsystem will process this feature list by building up a three dimensional model using tools in MFGMAP.

Note that in this case the feature list was originally defined by an upstream process and passed down to the planner. The planner at this point is not given the authority to delete this feature list without a request from the upstream processes.

```
Planner:    (Define   ((Name          UP_Stock_Geometry)
                      (Type          Block)
                      (Width         2.1 inches)
                      (Height        2.1)
                      (Length        2.1)
                      (P_Vector      (0.0  0.0  0.0))
                      (D_Vector      (0.0  2.1  0.0))
                      (W_Vector      (0.0 0.0 2.1))
                      (L_Vector      (2.1 0.0 0.0))
                      (Is_Part       UP_Stock_Part))

                      ((Name         UP_Stock_Part)
                      (Type          Part)
                      (Units         inches)
                      (Material      Aluminum)
                      (Source        <Batch #>)
                      (Surfaces      (Rolled Sawed Rolled Rolled Sawed Rolled)))
                      (Presentation  (1 2 3))
                      (Has_Parts     UP_Stock_Geometry))

                      ((Name         PL_Init_Stock)
                      (Type          Message)
                      (From          Planner)
                      (To            (Manager Cutting Holding Sensing)))
            )
```

We can see from this first message that there are several desirable system features. First, *optional* and *required* arguments must be handled gracefully, and secondly that *default* values can be used at any time. In this case, "inches" are specified by the "UP_Stock_Part" feature list and the units are essentially inherited from this. Finally, it should be noted that the definition of the stock material was defined in an upstream process, i.e., was input to the IMW.

Now the answers start coming back from the different subsystems. In cases of simple data transfer, there is not a need to represent the active state of verbs. A process has the option of sending a completed message, i.e., past tense, immediately. However, some confirmation must be sent back to the source of a message for every feature list in the message, except elements of type "message." Also the messages can be received in any order.

Cutting:
```
(Defined UP_Stock_Geometry
         UP_Stock_Part
         ((Reply_To   PL_Init_Stock)
         (Type        Message)
         (From        Cutting)
         (To          Planner)))
```

Manager:
```
(Defined UP_Stock_Geometry
         UP_Stock_Part
         ((Reply_To   PL_Init_Stock)
          (Type       Message)
         (From        Manager)
         (To          Planner)))
```

Sensing:
```
(Defined UP_Stock_Geometry
         UP_Stock_Part
         ((Reply_To   PL_Init_Stock)
         (Type        Message)
         (From        Sensing)
         (To          Planner)))
```

Holding:
```
(Defined UP_Stock_Geometry
         UP_Stock_Part
         ((Reply_To   PL_Init_Stock)
         (Type        Message)
         (From        Holding)
         (To          Planner)))
```

Naming every mail message can be annoying, however, there is a strong need for accountability. Therefore, it is a good convention to name original mail messages. Responses can then invoke the original name (e.g., with Reply_To) and thus it avoids unnecessary new names. At the same time, this convention simplifies the naming and management of reply mail.

A future refinement will be to find a more concise way to fully represent all of the message oriented information.

17. Modeling The Part

In this example, we are going to define a part that is essentially a squared block with one cut hole, which will be cut from the previously defined stock part. Note that in these definitions the geometry Is_Part *part* and is not an *operation*. Again, this feature list will be used to drive the primitive functions of MFGMAP in order to make a full three dimensional model.

```
Planning:   (Define   ((Name       UP_Geometry)
                       (Type       Block)
                       (Width      2.0)
                       (Height     2.0)
                       (Length     2.0)
                       (P_Vector   (0.0  0.0  0.0))
                       (D_Vector   (0.0  2.0  0.0))
                       (W_Vector   (0.0 0.0 2.0))
                       (L_Vector   (2.0 0.0 0.0))
                       (Has_Parts  UP_Center_Hole)
                       (Is_Part    (UP_Stock_Part  UP_Stock_Geometry)))

                      ((Name       UP_Center_Hole)
                       (Type       Thru_Hole)
                       (Radius     .25)
                       (Depth      2.0)
                       (P_Vector   (1.0 2.0 1.0))
                       (D_Vector   (1.0 0.0 1.0))
                       (Is_Part    UP_Surface))

                      ((Name       PL_Part)
                       (Type       Message)
                       (From       Planner)
                       (To         (Manager Cutting Holding Sensing)))
            )

Manager:    (Defined UP_Geometry UP_Center_Hole UP_Part ... )
Cutting:    (Defined UP_Geometry UP_Center_Hole UP_Part ... )
Sensing:    (Defined UP_Geometry UP_Center_Hole UP_Part ... )
Holding:    (Defined UP_Geometry UP_Center_Hole UP_Part ... )
```

18. Negotiation Example

In this example, the planner sends out messages to every subsystem at the same time. This will cause the subsystems to develop plans that are independent of each other and will give the planner a chance to study the conflicts. An optional scenario would be to send each of the subsystems a message in sequence, while imposing successively stronger constraints on each system in the list. While, this organization may seem simpler than the parallel strategy, it will not always converge on the best solution. For example, a solution proposed first by the cutting subsystem may unnecessarily over constrain the holding subsystem, which could result in a second rate solution for holding even though the cutting system was ambivalent.

```
Planner:    (Offer    ((Name      PL_Center_Hole_Op
                        (Type      Thru_Hole)
                        (Is_Part   (Operation UP_Center_Hole))

                       ((Name      PL_Offer1)
                        (Type      Message)
                        (From      Planner)
                        (To        (Manager Cutting Holding Sensing))
            )

Cutting:    (Offering ((Name      CX_Center_Hole_Op
                        (Type      Operation)
                        (Tools     (Drill2))
                        (NC_Name   Peck_Hole)
                        (Speed     20)
                        (Feed      20))

                       ((Reply_To  PL_Offer1)
                        (From      Cutting)
                        (To        Planner)))


Holding:    (Offering ((Name      CX_Center_Hole_Op
                        (Type      Operation)
                        (Fixture   (Fixed)
                        (Top_Face  4)
                        ((Reply_To PL_Offer1)
                        (From      Holding)
                        (To        (Planner Manager)))
```

At this point in the processing, the sensing system could respond with potential problems. For example, without yet knowing about the cutter, it is appropriate to worry about tool wear and tool breakage. It is also appropriate to worry about potential clogging of the drill which is often a problem when cutting aluminum. Finally, the surface finish around the hole might have burrs that have been "pulled out" from the hole. These problems can be addressed in a number of ways, including using a different drill and adding more steps to the process, e.g., a small hole chamfer. Of course, as concerns, they can be ignored.

```
Sensing:    (Offering ((Name       CX_Center_Hole_Op
                       (Type       Operation)
                       (Concerns   (Tool_Breakage
                                   Surface_Burrs
                                   Chip_Clogs))

                       ((Reply_To  PL_Offer1)
                       (From       Sensing)
                       (To         (Planner Manager)))

Manager:    (Offering ((Name       CX_Center_Hole_Op
                       (Type       Operation)

                       ((Reply_To  PL_Offer1)
                       (From       Manager)
                       (To         Planner)))
```

19. Setup Example

```
Planner:    (Make       UP_Part
                        ((Name      PL_Workpackage)
                        (Type       Message)
                        (From       Planner)
                        (To         Manager)))

Manager:    (Making     UP_Part
                        ((Reply_To  PL_Workpackage)
                        (Type       Message)
                        (From       Manager)
                        (To         Planner)))

Manager:    (Make       HX_Setup1
                        ((Name      PM_Setup1)
                        (Type       Message)
                        (From       Manager)
                        (To         (Holding Sensing)))

Holding:    (Making     HX_Setup1
                        ((Reply_To  PM_Setup1)
                        (Type       Message)
                        (From       Holding)
                        (To         Manager)))

Sensing:    (Get        HX_Setup1
                        ((Reply_To  SX_Get_Setup1)
                        (Type       Message)
                        (From       Sensing)
                        (To         Holding)))

Holding:    (Got        ((Name      HX_Setup1)
                        ...)

                        ((Type      Message)
                        (Reply_To   SX_Get_Setup1)
                        (From       Holding)
                        (To         Sensing)))

Sensing:    (Monitoring HX_Setup1
                        ((Type      Message)
                        (Reply_To   PM_Setup1)
                        (From       Sensing)
                        (To         Manager)))

Holding:    (Made       HX_Setup1
```

```
                ((Type          Message)
                (Reply_To       PM_Setup1)
                (From           Holding)
                (To             Manager)))

Sensing:        (Monitored HX_Setup1
                ((Type          Message)
                (Reply_To       PM_Setup1)
                (From           Sensing)
                (To             Manager)))
```

20. Machining Example: Part Completed

```
Manager:        (Made   UP_Part
                ( (Reply_To     PL_Workpackage)
                (Type           Message)
                (From           Manager)
                (To             Planner)))
```

After a part has been accepted by the upstream process, it then issues authorization to delete the stock and part descriptions, which is then passed on to the various subsystems.

```
Upstream:       (Delete UP_Stock_Part
                UP_Stock_Geometry
                UP_Finished_Part
                UP_Finished_Geometry
                UP_Center_Hole

                ((Name          UP_Completed)
                (Type           Message)
                (To             Planner)
                (From           Upstream)))

Planner:        (Deleted UP_Stock_Part
                UP_Stock_Geometry
                UP_Finished_Part
                UP_Finished_Geometry
                UP_Center_Hole

                ( (Reply_To:    UP_Completed)
                (Type           Message)
                (To             Upstream)
                (From           Planner)))
```

This example suggests "wildcards" in name could be very useful, such as UP_*, which would mean that all feature lists that start with "UP_" should be deleted.

Center for
Integrated Manufacturing Decision Systems

FEL Interface to the Planning Expert

Paul Erion

March, 1990

Abstract:

This document describes the detailed syntax of the FEL (Feature Exchange Language) sentences that are understood (and generated) by the process planner used in the IMW (Intelligent Machining Workstation) prototype system.

A description of the Planner's interaction with the modeler is given, together with a detailed explanation of the meaning of the feature lists received from the modeler. The method of requesting that the Planner produce a plan and the FEL sentence that it produces as a response to such a request are explained in detail.

Copyright © 1990 Carnegie Mellon University

Contact:

David Bourne
CIMDS
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-8810

FEL Interface to the Planning Expert

This section assumes that the reader has some familiarity with: FEL syntax, the MACHINIST[1], and the Modeler[2].

The Planning Expert (PX) is based on the MACHINIST, a program designed and written by Caroline Hayes. The MACHINIST is a process planner that produces manufacturing plans for machined parts. PX is simply the integration of the MACHINIST and the Lisp-based Generic Expert. This integration allows input to, and output from, the MACHINIST to be via FEL sentences.

1. Input to PX

PX's function is to produce a manufacturing plan for a specified part. In order to accomplish this task, certain information is required by PX. The Process Planner needs a description of the stock envelope, the part envelope, and the features that comprise the part. The Modeler is the expert that will be queried for that information.

1.1. Creating the Model

In order for PX to plan for the machining of a part, the Modeler must have a model of the part and stock objects. The model of the part object will include a specification for each of the part's feature. The *ADD* sentence that follows is input to MX that would create a part object named *boss*, and a stock object[3] named *S0235*. The part and stock models created by this sentence will be referred to in subsequent examples.

```
(add    ( (name          G0927)
          (type          message)
          (to            mx) )

        ( (type          application)
          (name          planner) )

        ( (name          imw)
          (type          environment)
          (application   planner) )

        ( (name          S0235)
          (type          object)
          (material      aluminum)
          (p_vector      (0.0  0.0  0.0))
```

1. The term *Process Planner* will be used as a synonym for *MACHINIST*.
2. The terms *MX* and *Modeler* will be used interchangeably to refer to the Modeling Expert.
3. For a detailed explanation of the attributes used in the *ADD* sentence, see the section in this document entitled, *FEL Syntax for Communicating with a Geometric Modeler*.

```
   (w_vector      (3.5  0.0  0.0))
   (l_vector      (0.0  2.5  0.0))
   (d_vector      (0.0  0.0  1.5)) )

 ( (name          boss)
   (type          object)
   (material      aluminum)
   (p_vector      (0.0  0.0  0.0))
   (w_vector      (3.0  0.0  0.0))
   (l_vector      (0.0  2.0  0.0))
   (d_vector      (0.0  0.0  1.0)) )

 ( (name          thru1)
   (type          thru_hole)
   (object        boss)
   (p_vector      (1.5   1.0   1.0))
   (d_vector      (0.0   0.0  -1.0))
   (radius        0.3) ))
```

Figure 1 is a graphical representation of *boss*, the part object added to the Modeler.

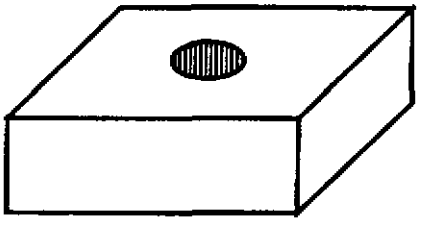


Figure 1. The part to be machined.

1.2. Querying the Modeler

MX may contain models for many objects. So, if PX is to construct a plan, it needs certain information in order to be able to locate the desired part and stock objects in the Modeler's hierarchy. That is, the requester of the manufacturing plan must provide the name of the application, the environment, the part, and the stock. Hence, the initial function of PX[4] expects an FEL sentence of the following form[5]:

```
(plan    ( (name          <name>)
           (type          planning_op)
           (application   <application-name>)
           (environment   <environment-name>)
```

4. The initial function is the function executed by the generic expert component of an expert when a sentence arrives that initiates a new dialogue.
5. Typically, example FEL sentences will not include the feature list of type *message*.

```
(part       <part-name>)
(stock      <stock-name>) ))
```

Currently, PX's initial function is only capable of performing meaningful actions for a `plan` sentence (that is, a sentence with the verb `plan`). The sentence should contain one feature list of type `planning_op`.[6] This feature list contains information that will allow PX to query MX. The values of the attributes `application`[7] and `environment` specify the location of the objects in the Modeler's hierarchy. The values of the attributes `part` and `stock` are the names of the objects about which the process planner needs feature information.

Once PX has extracted the names of the part and stock objects from the `plan` sentence, the next step is to query the Modeler for the features that are associated with both of these objects. Therefore two dialogues are initiated with MX. One of the dialogues requests the features of the part and, the other, requests the features of the stock[8].

Following are two sentences (a request and a reply) that represent a sample dialogue between PX and MX. The purpose of the exchange is for PX to obtain the names of the features associated with the part. The part is an object named `boss` that resides in the application, `planner`, and the environment, `imw`.

```
(get    ( (name          px_1)
          (type          message)
          (to            mx)
          (from          px) )

        ( (name          boss
          (type          object)
          (application   planner)
          (environment   imw) ))


(got    ( (name          px_1)
          (type          message)
          (to            px)
          (from          mx) )

        ( (name          boss
          (type          object)
          (application   planner)
          (environment   imw)
          (material      aluminum)
```

6. PX is not smart enough to be able to plan more than one part per dialogue.
7. The attribute `:application` is optional. If it is not provided, then the symbol `:planner` is used as the value for this attribute.
8. One's initial thought might be that the stock has no features; however, that is not the case. Associated with every object is a rectangular bounding box. This is a feature of the object and is known as the envelope of the object. Knowledge about the part and stock envelopes is very important to the MACHINIST.

```
                (feature      (envelope  hole)) ))
```

Once both of the dialogues with MX have terminated, the returned FEL sentences are dissected. At this point, each sentence contains two attribute/value pairs that are of interest to PX. The attributes of importance are `material` and `feature`.

The value of the attribute `material` specifies the material composition of the object[9]. This information is converted into a data structure that will be processed by the MACHINIST[10]. In the preceding example, the part will machined out of aluminum.

Associated with each object is a set of features. For each of these features, the Modeler contains detailed information. The process of acquiring that information from MX commences once the names of all of the features are known to PX. These names are the elements of the list that is the value of the attribute `feature`. These feature names will generate the next round of dialogues with the Modeler. The above example, would involve two distinct dialogues. One for each of the features named; that is, `envelope` and `hole`.

For each feature associated with an object, a `get` request will be sent to the Modeler (each of the requests initiating a new dialogue). The responses to these queries will contain information needed by the MACHINIST. As an example, if PX wanted to obtain modeling information about a feature named `hole`, associated with the object, `boss`, in the application, `planner`, and the environment, `imw`; then the following FEL sentence would be sent to MX.

```
(get    ( (name          px_2)
          (type          message)
          (to            mx)
          (from          px) )

        ( (name          hole
          (type          feature)
          (application   planner)
          (environment   imw) ))
```

The response from MX would take the form:

```
(got    ( (name          px_2)
          (type          message)
          (to            px)
          (from          mx) )

        ( (name          hole)
          (type          thru_hole)
          (application   planner)
```

9. It should come as no surprise that the part and stock objects should have the same value for the attribute `:material`.
10. The MACHINIST is an expert system that utilizes the rule-based programming language, OPS5. As a consequence, the data structures processed by the MACHINIST are Working Memory Elements.

```
(environment   imw)
(opens_on      (mx_7  mx_9))
(distance      ( (mx_11 1.5)  (mx_8 0.5) ))
(radius        0.3)
(depth         2.0) )).
```

Actually, the Modeler returns more information than is shown; but only a subset of the attribute/value pairs returned in the feature list of type `thru_hole` are needed by PX. MX is the modeling expert to all experts. It is the duty of the expert requesting information to parse the FEL sentence and extract the attribute/value pairs that make sense to it.

Initially, PX only knows the name of the feature; in this case `hole`. Once the preceding `got` sentence is received, PX can determine the feature type of `hole`. First, the feature list that contains the attribute/value pair, `(name hole)`, is located[11]. From that feature list, the value of the attribute `type` will provide the type of feature. In the current example, the value,`thru_hole`, denotes that the feature is a through hole.

When PX knows the type of feature, then it knows which elements (that is, attribute/value pairs) to extract from the feature list. The attribute/value pairs arrive in a form that the MACHINIST is not capable of processing. Therefore, steps are taken to transform the "raw" information into the data structures used by the MACHINIST. Since the Process Planner is implemented in OPS5, these data structures are Working Memory Elements.

For example, the attribute/value pair:

```
(distance      ((mx_11 1.5)  (mx_8 0.5)))
```

is converted into the following Working Memory Elements:

```
(center_dist   ^of      hole
               ^from    <name used by MACHINIST for mx_11>
               ^is      1.5
               ^status  filled)

(center_dist   ^of      hole
               ^from    <name used by MACHINIST for mx_8>
               ^is      0.5
               ^status  filled).
```

11. It is possible, though not likely, for the dialogue to have the same name as the feature. In other words, the feature list of type `:message` and the feature list of type `:thru_hole` have the same value for the attribute `:name`. PX takes this into account, by ignoring the feature list of type `:message`.

1.3. Feature Lists for Features: the Attribute/Value Pairs of Interest

From the perspective of the Process Planner, the Modeler can currently provide information for the features[12]: blind_hole, envelope, thru_ angle, thru_hole, and thru_slot. The following sections cover those five features. Specifically, the attribute/value pairs of interest to PX are listed and explained for the feature in question.

1.3.1. Blind_Hole

In a feature list of type `blind_hole`, the attribute/value pairs of interest to PX are those with the attributes:

depth The value of this attribute is a real number which gives the depth of the hole.

distance The value of this attribute specifies the center of the hole. The value is of the form:

$$((\text{side}_i\ \text{distance}_i)\ (\text{side}_j\ \text{distance}_j)).$$

This gives the hole's center by providing the distance of the center of the hole from two orthogonal sides of the envelope of the part object.

opens_on This attribute's value is a list of one element, a face name from the set of face names assigned to the part envelope by MX. The face name denotes the face upon which the hole opens.

radius The value of this attribute is a real number that gives the radius of the blind hole.

1.3.2. Envelope

The envelope describes a rectangular bounding box around an object. Remember, the MACHINIST expects two objects, the part and the stock. Associated with each of these objects is an envelope feature.

The Process Planner and the Modeler use different naming schemes for surface finishes and face names. But experts who wish to converse with PX, should not be concerned with the naming conventions used by MACHINIST. Hence, mapping functions are employed by PX to translate between the differing, naming schemes.

12. The Modeler can model more than these five features. However, the other features are currently described in terms of vectors. Since the Process Planner is feature based, information concerning vectors is of limited utility.

For the surface finishes, the correspondence between the two schemes is known *a priori*. Therefore, that mapping function is created at system start-up. However, that is not the case for face names. The names attached by MX to the face of an envelope are dynamic; that is, they are created when the model is added to the Modeler. Hence, PX uses the information provided in the envelope feature list to set up a mapping function between the names used by the Modeler and the names employed by MACHINIST. The result is that when experts converse with PX, they do not have to be concerned with how the Process Planner represents face names or surface finishes. A single representation can be used, that of the Modeler.

In a feature list of type `envelope`, the attribute/value pairs of interest to PX are those with the attributes:

`distance` The value of this attribute specifies the dimensions of the envelope. The value is of the form:

$$((side_1\ side_2\ distance_1) \\ (side_3\ side_4\ distance_2) \\ (side_5\ side_6\ distance_3)).$$

Each element of the list specifies the distance between two parallel faces of the envelope. Let X be such an element. The first two elements of X are face names from the set of face names assigned to the part envelope by MX, with the added constraint that the faces be parallel to one another. The third element, a real number, is the distance between these two faces. So, if X is equal to:

$$(side_3\ side_4\ distance_2)$$

then $side_3$ is parallel to $side_4$ and $distance_2$ is the distance separating these two faces.

`finish` This attribute's value is a list of the form:

$$((side_1\ finish_1)\ \ldots\ (side_6\ finish_6)).$$

$side_i$ (where i = 1 to 6) is a face name from the set of face names assigned to the part envelope by MX, and $finish_i$ is the surface finish of that face. $finish_i$ may be assigned one of the values: `none`[13], `machined`, `rolled`, or `saw_cut`.

13. If the Modeler has assigned a value of *none* to a face, then PX assumes the face is saw cut. The value of `saw_cut` is chosen, since that presents a worst case scenario.

normal The value of this attribute is a list of the form:

((side$_1$ normal$_1$) ... (side$_6$ normal$_6$)).

side$_i$ (where i = 1 to 6) is a face name from the set of face names assigned to the part envelope by MX. normal$_i$ is the unit normal vector for that face.

1.3.3. Thru_Angle

In a feature list of type thru_angle, the attribute/value pairs of interest to PX are those with the attributes:

angle This attribute's value is a list of two elements. The first element is a face name from the set of face names assigned to the part envelope by MX. The second element is a real number between 0 and 90, exclusive. The second element is the degree of the angle formed by the new face created by the feature and the face given by the first element. In Figure 2, for example, the value of the attribute, angle, could be either (B θ), or (E ϕ); where $\phi = (90 - \theta)$.

distance The value of this attribute gives the distance from one of the faces not affected by the thru angle to a vertex of an angle formed by the feature. The value is of the form:

((side$_i$ distance$_i$)).

side$_i$ is the name of a face not affected by the thru angle. Consequently, it will not be an element of the value of the attribute, opens_on. distance$_i$ is the distance along an orthogonal face to a vertex of the angle formed by the thru angle. For example, in Figure 2, this attribute's value could be either ((D δ)) or ((F ω)).

opens_on This attribute's value is a list of four elements, all are face names from the set of face names assigned to the part envelope by MX. The four faces are those that the thru angle feature will open out upon. That is, if we envision the tool making this feature, these faces will have the tool pass through them. In Figure 2, the thru angle opens out upon the faces: A, B, C, and E.

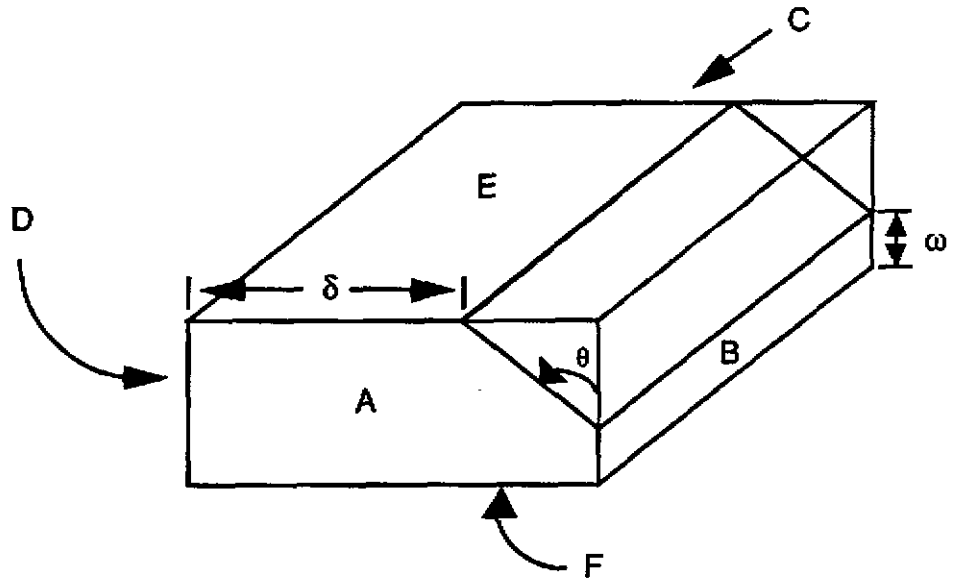


Figure 2. A block with a thru angle feature.

1.3.4. Thru_Hole

In a feature list of type `thru_hole`, the attributes whose values are meaningful to PX are:

depth The value of this attribute is a real number which gives the depth of the hole.

distance The value of this attribute specifies the center of the hole. The value is of the form:

$$((\text{side}_i\ \text{distance}_i)\ (\text{side}_j\ \text{distance}_j)).$$

This gives the hole's center by providing the distance of the center of the hole from two orthogonal sides of the envelope of the part object.

opens_on This attribute's value is a list of two elements, both are face names from the set of face names assigned to the part envelope by MX. These two faces are the opposite sides of the envelope upon which the hole opens.

radius The value of this attribute is a real number that gives the radius of the thru hole.

1.3.5. Thru_Slot

In a feature list of type `thru_slot`, the attribute/value pairs of interest to PX are those with the attributes:

depth This attribute's value is a real number that gives the depth of the slot from the part envelope. In Figure 3, this would be ϕ.

distance The value of this attribute gives the distance from a face, to the closest edge of the thru slot. The value is of the form:

$$((side_i\ distance_i)).$$

$side_i$ is the name of a face that meets two conditions: (i) it is not affected by the thru angle, and (ii) the face is not parallel to the bottom of the slot. $distance_i$ is the distance from $side_i$ to the closet edge of the thru slot. For Figure 3, an example of a value for `distance` is: `((A α))`

opens_on This attribute's value is a list of three elements, all are face names from the set of face names assigned to the part envelope by MX. The three faces are those that the thru slot feature will open out on. That is, if we envision the tool making the feature, these faces will have the tool pass through them.

width This attribute's value is a real number that gives the width of the slot. This is the value, β, in Figure 3.

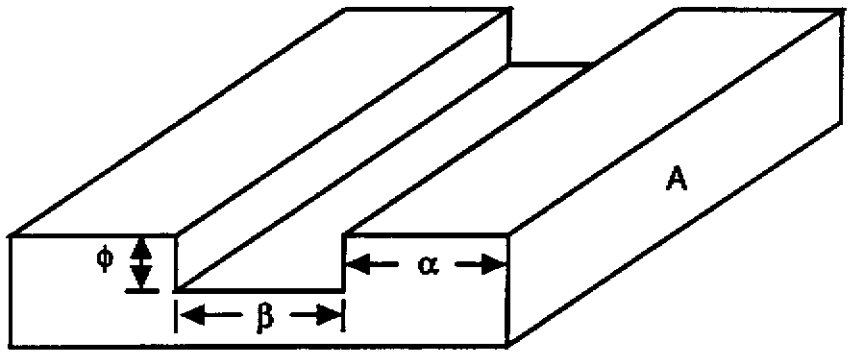


Figure 3. A block with a thru slot feature.

2. Output from PX

The MACHINIST is run once the information from the Modeler, that describes the part and stock objects, has been received and transformed into OPS5 Working Memory Elements. Originally, the result of a run of the MACHINIST was simply displayed to the terminal. This is of little use to an expert that requests a machining process plan. Hence, part of the function of PX is to gather the plan together and construct an FEL sentence that embodies the plan. This sentence will be returned as the response to the original *PLAN* request.

Each of the next three subsections is an example that will expand upon a portion of the preceding paragraph. Following is a brief description of the examples to be used.

(1) Presented is an example of a *PLAN* sentence sent to PX, requesting a process plan for the specified part.

(2) The process plan is presented as it would have been originally output to the terminal by MACHINIST[14]. This plan is not in the form of an FEL sentence.

(3) Shown is the *PLANNED* sentence returned in response to the *PLAN* request. This sentence will contain the process plan transformed into a meaningful form (that is, feature lists of attribute/value pairs).

2.1. Requesting a Machining Plan

If the Human Interface (HI) wanted a process plan for machining the part, *boss*, from the piece of stock, *S0235*; then HI would need to construct, and send to PX, an FEL sentence of the form:

```
(plan   ( (name          G0942)
          (type          message)
          (to            px)
          (from          hi) )

        ( (name          plan_boss)
          (type          planning_op)
          (application   planner)
          (environment   imw)
          (part          boss)
          (stock         S0235) )).
```

PX does not need detailed information about the part and stock models from the expert requesting a machining process plan. All that is required is the names of the part and stock objects, and their location in the modeling hierarchy. The name of the part object is the value of the attribute, `part`, and the name of the stock object is the value of

14. Excluding some inessential graphics.

the attribute, `stock`. The location of these objects in the modeling hierarchy is given by the values of the attributes: `application` and `environment`.

2.2. Output Displayed by the Process Planner

The previous example was of a *PLAN* sentence directed to PX. The response returned by PX is solely determined by the outcome of the Process Planner. If it is not possible to plan the part, the response will be a *NOTPLANNED* sentence. When the Process Planner is able to plan machining operations for the part, then a *PLANNED* sentence is returned. The contents of this sentence are based upon the plan output to the terminal by MACHINIST. For the part currently under consideration, MACHINIST produces the plan:

```
THE STOCK DIMENSIONS:
        3.5 x 2.5 x 1.5

THE PLAN:

        Set-Up #1, use VISE
             - Put side 1 UP
             - Put side 4 DOWN
             - Put side 5 ON_SOLID_JAW
             - FACE_MILL 1


        Set-Up #2, use Vise
             - Put side 2 UP
             - Put side 1 ON_SOLID_JAW
             - Put side 5 DOWN
             - FACE_MILL 2 to size


        Set-Up #3, use Vise
             - Put side 4 UP
             - Put side 1 DOWN
             - Put side 2 ON_SOLID_JAW
             - END_MILL 3 to size
             - FACE_MILL 4
             - DRILL THRU1
```

2.2.1. Explanation of Output

The process plan employs three setups to machine the part, `boss`. An individual setup provides (a) the fixturing device to use, (b) the orientation of the stock piece with respect to the bed of the tool table and the fixturing device, and (c) the features to be machined.

In the preceding plan, a vise is the fixturing device utilized in all three of the setups. A sine table would be an example of another device.

In order to cut features, the stock must be placed on the tool table. For an individual setup, the process plan dictates how to orient the sides of the stock. For example, in setup #2, side 5 is placed face down on the tool bed (`Put side 5 DOWN`), and side 1 is placed against the solid jaw of the vise (`Put side 1 ON_SOLID_JAW`). Directions are also given for orienting side 2; however, that information may be ignored, since side 2 is parallel to side 5.

2.3. PX's Response to *PLAN* Sentence

From the information culled from the process plan generated by the MACHINIST component of PX, a *PLANNED* sentence is constructed and returned to the Human Interface (HI).

2.3.1. Feature List Ordering for Setups and Machined Features

The order of the feature lists in the *PLANNED* sentence is critical. It is the ordering of the feature lists that determines, both, the ordering of the setups and the ordering of the features machined during each setup. That is, when iterating through the feature lists of the *PLANNED* sentence, the setups are executed in the order in which they are read. This method of ordering ("executing" the feature list as it read) also applies to the feature lists whose types designate a feature to be machined. Hopefully, the following example will clear up any confusion. In the *PLANNED* sentence (in Section 2.3.5), `setup_47` is executed first, followed by `setup_49`, then `setup_51`. During `setup_51`, feature `face_52` is machined, followed by `face_53`, and finally, `thru1`.

2.3.2. *PLANNING_OP* Feature List

The above discussion concentrated on the feature lists that pertained to setups and machined features, but one feature list precedes all of these. The feature list in question is of type `planning_op`. In this feature list, the attribute, `translation`, is the only one whose value contains useful information. The value of this attribute, a vector[15], provides the X, Y, and Z offsets necessary to correctly translate the part object within the stock object. What is meant by a correct translation? If all of the face features, given in the process plan, are removed from the stock object, then the vertices of the part envelope will be equal to the vertices of the modified stock object.

15. A vector is a list with exactly three real numeric elements. In the case of translation, the elements of the vector specify the translation in X, Y, and Z, respectively.

2.3.3. *SET_UP* Feature List

The primary function of a feature list of type `setup` is (a) to give the basic fixture that will be employed, and (b) provide information that will allow other experts to orient the stock with respect to that fixture and any other fixtures[16] employed.

2.3.3.1. Attributes of Generic *SET_UP* Feature List

A generic `setup` feature list consists of the attributes (ignoring `name` and `type`):

`method` A symbol used to designate the type of fixture to be used in this setup. Typical values include: `vise`, `angle_plate`, `sine_table`, `subplate`, or `toe_clamps`.

`major-ref` The attribute's value is a face name from the set of face names assigned to the stock by MX. The face name denotes the face that the Process Planner has designated as the major reference side. This is important for probing operations.

`minor-ref` The attribute's value is a face name from the set of face names assigned to the stock by MX. The face name denotes the face that the Process Planner has designated as the minor reference side. This is important for probing operations.

`major-pos` The value of this attribute is used in conjunction with the values of the attributes `major_ref` and `method`. It specifies the orientation of the face named by `major_ref` with respect to the fixture designated by `method`. For example, the attribute/value pairs:

```
(method      vise)
(major_ref   mx_5)
(major_pos   on_solid_jaw)
```

state that face `mx_5` is to be placed against the solid jaw of the vise.

`minor_pos` The value of this attribute is used in conjunction with the values of the attributes `minor_ref` and `method`. It specifies the orientation of the face named by `minor_ref` with respect to the fixture designated by `method`. For example, the attribute/value pairs:

```
(method      vise)
```

16. In this context, the term *fixtures* is being used loosely. In addition to what one normally considers a fixture, include the tool bed, a sine table, or a subplate.

```
(minor_ref    mx_3)
(minor_pos    down)
```

state that face mx_3 is to be placed face down in the vise.

major_normal This attribute's value is a vector. Specifically, it is the unit normal vector of the face denoted by the value of the attribute, major_ref.

minor_normal The value of this attribute is a vector. Specifically, it is the unit normal vector of the face denoted by the value of the attribute, minor_ref.

x_rotation A real number that specifies the number of degrees the model must be rotated[17] about the x-axis; such that, the appropriate face[18] has the unit normal vector, (0 0 1).

y_rotation A real number that specifies the number of degrees the model must be rotated[17] about the y-axis; such that, the appropriate face[18] has the unit normal vector, (0 0 1).

z_rotation The current version of PX always considers (0 0 1) to be the desired unit normal vector. Therefore, no matter what the initial orientation of the model, a rotation about the z-axis can never bring about the desired orientation. Consequently, The value of this attribute will always be 0.

The astute reader may have noticed that two of the attributes given for the feature list of type setup, major_ref and minor_ref, do not appear to be deducible from the output of the MACHINIST. This information is not concocted by PX. It is internal to the Process Planner, but is never displayed to the user. However, other experts do require this information; consequently, it is collected by PX, and added to the setup feature list.

---

17. The convention is adopted that positive rotations are such that, when looking from a positive axis toward the origin, a ninety degree counterclockwise rotation will transform one positive axis into the other. The following table, usable for either right-handed or left-handed coordinate systems, derives from this convention:

Axis of Rotation	Direction of Positive Rotation
X	Y --> Z
Y	Z --> X
Z	X --> Y

18. By convention, a face whose unit normal vector is (0 0 1) is considered to face up. In determining which face should be up, the first step is to examine the values of the two attributes major_pos and minor_pos, and note which one has a value of either up or down. Next, find the related "reference" attribute. Remember, a relation exists between the values of the attributes major_ref and major_pos, and also between the attributes minor_ref and minor_pos. Finally, if the value of the "position" attribute is up, then the value of the corresponding "reference" attribute is the name of the face to be placed up. If the value of the "position" attribute is down, then the face opposite to the

2.3.3.2. Additional Attributes for Setup Employing a Sine Table

The preceding attributes were described as belonging to a generic `setup` feature list. In other words, those attributes will be found in all feature lists of type `setup`, regardless of the value of the attribute `method`. However, if `sine_table` is the value of method, then there are two additional attribute/value pairs:

`angle`	The value of this attribute is a real number which provides the degree of the angle that should exist between the sine table and the tool bed.
`position`	This attribute's value is one of the three symbols: `X`, `Y` or `Z`. The symbol denotes the axis to which the hinge of the sine table is parallel.

2.3.4. Feature Lists for Machined Features

When constructing the *PLANNED* sentence, PX views machined features as falling into two classes. Features that are originally added to the model (e.g., a thru hole) and ones that are added by the Process Planner.

2.3.4.1. Original Features of the Part Model

The first class, features originally added to the model, are trivially handled by PX. Adding a feature list that describes machining features of this class, requires no more effort than giving the name and type of the feature. Since the MACHINIST does not alter anything that would affect MX's definition of the feature, there is no need to provide any other information. For example, to describe the thru hole, `thru1`, the feature list:

```
( (name  thru1)  (type  thru_hole))
```

would suffice.

---

value of the corresponding "reference" attribute is the face to be oriented up. For example, given the attribute/value pairs:

```
(major_ref  mx_5)
(minor_ref  mx_3)
(major_pos  on_solid_jaw)
(minor_pos  down)
```

the face opposite to `mx_3` will be the face to be placed up, since the value of `minor_pos` is `down`.

2.3.4.2. Face Features Added by Process Planner

Features added by the Process Planner comprise the second class. The current version of MACHINIST generates new features during the process of reducing the stock envelope to the part envelope (i.e., squaring the stock). To reduce the stock, MACHINIST specifies milling operations. The Process Planner distinguishes between two types of milling operations, face mills and end mills. However, the Modeler does not distinguish between the two; they are both simply defined as face features (see Figure 4).

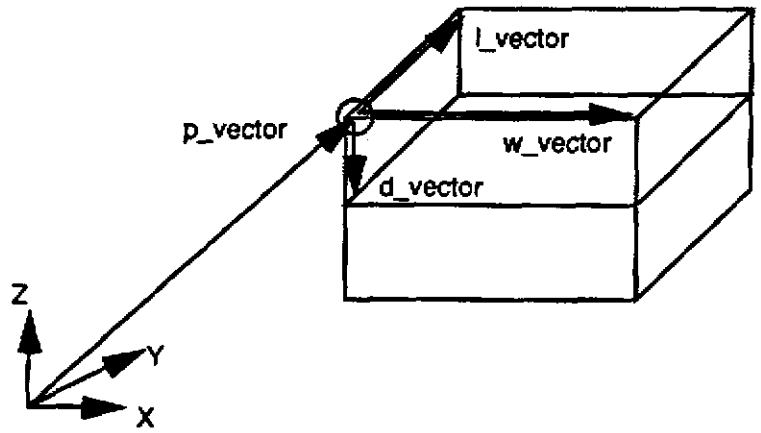


Figure 4. Definition of a Face Feature.

When PX detects one of the milling operations, a feature list of type `face` is added to the *PLANNED* sentence. This feature list should contain the information necessary for MX to model the feature. Therefore, ignoring `name` and `type`, a `face` feature list consists of the attributes:

`face`	The attribute's value is a face name from the set of face names assigned to the stock by MX. The face name denotes the side upon which the milling operation will take place.
`p_vector`	The value of this attribute is a vector. The origin is the model origin. The vector points to an open corner of the face that is to be machined away.
`l_vector`	The value of this attribute is a vector. The origin is defined by the value of the attribute, `p_vector`. The vector points along an open edge of the area to be milled away.
`w_vector`	This attribute's value is a vector. The origin is defined by the value of the `p_vector` attribute. The vector points along an open edge of the area to be machined away.

d_vector This attribute's value is a vector. The origin is defined by the value of the attribute, p_vector. The vector points along an open edge of the area to be machined away. However, this vector is directionally significant. It must point toward the part face.

2.3.5. *PLANNED* Sentence for the Part, *BOSS*

The following *PLANNED* sentence is the FEL translation of the process plan generated by MACHINIST for the part, *boss*. This is also the reply that would be generated for our example *PLAN* sentence.

```
(planned ( (name            G0942)
           (type            message)
           (to              hi)
           (from            px) )

         ( (name            plan_boss)
           (type            planning_op)
           (application     planner)
           (environment     imw)
           (part            boss)
           (stock           S0235)
           (translation     (0.0  0.5  0.25)) )

         ( (name            setup_47)
           (type            setup)
           (method          vise)
           (major_ref       mx_6)
           (minor_ref       mx_3)
           (major_pos       down)
           (minor_pos       on_solid_jaw)
           (major_normal    (0.0   0.0  -1.0))
           (minor_normal    (0.0   1.0   0.0))
           (x_rotation      0)
           (y_rotation      0)
           (z_rotation      0) )

         ( (name            face_48)
           (type            face)
           (face            mx_5)
           (p_vector        ( 3.5   2.5   1.5))
           (l_vector        (-3.5   0.0   0.0))
           (w_vector        ( 0.0  -2.5   0.0))
           (d_vector        ( 0.0   0.0  -0.25)) )

         ( (name            setup_49)
           (type            setup)
           (method          vise)
```

```
  (major_ref      mx_5)
  (minor_ref      mx_3)
  (major_pos      on_solid_jaw)
  (minor_pos      down)
  (major_normal   (0.0  0.0  1.0))
  (minor_normal   (0.0  1.0  0.0))
  (x_rotation     -90)
  (y_rotation       0)
  (z_rotation       0) )

( (name           face_50)
  (type           face)
  (face           mx_1)
  (p_vector       ( 3.5  0.0   1.5))
  (l_vector       (-3.5  0.0   0.0))
  (w_vector       ( 0.0  0.0  -1.5))
  (d_vector       ( 0.0  0.5   0.0)) )

( (name           setup_51)
  (type           setup)
  (method         vise)
  (major_ref      mx_5)
  (minor_ref      mx_1)
  (major_pos      down)
  (minor_pos      on_solid_jaw)
  (major_normal   (0.0   0.0  1.0))
  (minor_normal   (0.0  -1.0  0.0))
  (x_rotation     180)
  (y_rotation       0)
  (z_rotation       0) )

( (name           face_52)
  (type           face)
  (face           mx_6)
  (p_vector       ( 3.5   2.5  0.0))
  (l_vector       ( 0.0  -2.5  0.0))
  (w_vector       (-3.5   0.0  0.0))
  (d_vector       ( 0.0   0.0  0.25)) )

( (name           face_53)
  (type           face)
  (face           mx_2)
  (p_vector       ( 3.5  0.0  0.0))
  (l_vector       ( 0.0  2.5  0.0))
  (w_vector       ( 0.0  0.0  1.5))
  (d_vector       (-0.5  0.5  0.0)) )

( (name           thru1)
  (type           thru_hole) ))
```

Center for
Integrated Manufacturing Decision Systems

FEL Syntax for Communicating with a Geometric Modeler

Duane T. Williams

March, 1990

Abstract:

This document describes the detailed syntax of the FEL (Feature Exchange Language) sentences that are understood (and generated) by the geometric modeler used in the IMW (Intelligent Machining Workstation) prototype system.

A brief description of the general form of FEL sentences is given, as well as a brief overview of the hierarchical structure of objects that the modeler is capable of representing. The bulk of the document describes the details of sentences that cause models composed of objects to be constructed, that cause features to be added to objects, that enable information about models to be retrieved by remote processes, and that enable models to be transformed and graphically displayed.

Copyright © 1990 Carnegie Mellon University

Contact:
David Bourne
CIMDS
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-8810

This document describes the syntax of FEL sentences understood by the Modeling Expert (MX). It also explains the interface between the modeler and the program (MXD) that displays models on the Sun.

1. General FEL Syntax

The usual syntax of FEL sentences is a parenthesized list whose first element is a verb and whose subsequent elements are feature lists.

(**aVerb** featureList1 ... featureListN)

A feature list is a parenthesized list of attribute-value pairs. An attribute-value pair is a parenthesized list of two elements: an attribute name and a value.

((attribute1 value1) ... (attributeN valueN))

In general, an FEL sentence should contain a feature list that names the sentence and specifies its source and destination (usually the names of experts, e.g., PL, CX, MX, etc.). This feature list must contain a TYPE attribute with value MESSAGE. Thus, FEL sentences will usually have the following form:

```
( aVerb  ( (type message) (name aName) (from aSource) (to aDestination) )
         featureList2
         ...
         featureListN
)
```

Each of the other feature lists in a sentence should be a complete and independent packet of information for a single application of the verb. In other words, the sentence above should be equivalent to the following sequence of sentences:

```
( aVerb  ( (type message) (name aName) (from aSource) (to aDestination) )
         featureList2
)

   ...

( aVerb  ( (type message) (name aName) (from aSource) (to aDestination) )
         featureListN
)
```

This condition of feature list independence is not an absolute requirement, but it is highly desirable, because it simplifies the interpretation of individual sentences.

It is the primary purpose of this document to describe the verbs and feature lists that are meaningful to the modeler, and the information that it is currently capable of providing.

2. Modeler Hierarchy

The basic thing to know about the modeler is that it organizes geometry in a hierarchy: worlds, applications, environments, objects, features, and faces (see Figure 1 on page 54). Worlds, applications, environments, objects, and features are collections of the elements at the next level down in the hierarchy. Each *object* is associated with a TWIN[1] BREP structure that approximates the intended geometry of the object. Each *face* contains a connection with the corresponding BREP structure that represents its geometry.

Every element of the hierarchy is given a name, which must be unique within the collection of which the element is a part. Thus, the names of objects within a particular environment must be unique, but the names may be reused in other environments, or at other levels of the hierarchy. In order for an FEL sentence to refer to a particular node in the hierarchy, it must specify the path through the hierarchy that leads to that node. For example, in order for an FEL sentence to refer to a particular *object*, it must specify the world, the application, the environment and the object. This is done by giving the name of each of these things.

In the current implementation of the modeler, there is only one world and it is never named explicitly in FEL sentences. All applications are automatically part of this default world. Furthermore, there is a predefined application named "DEFAULT_APP" and it contains a predefined environment named "DEFAULT_ENV"; so it is possible to construct a simple model without explicit mention of either an application or an environment.

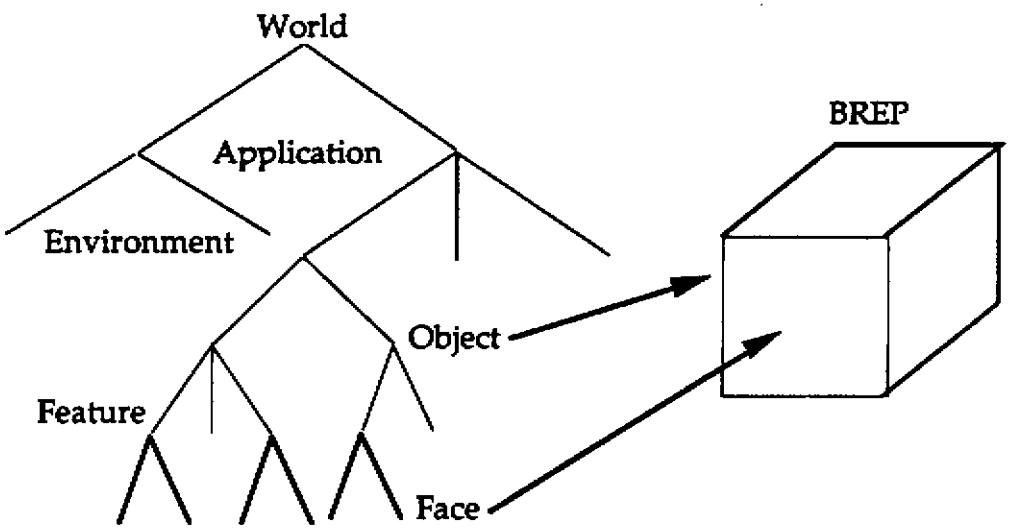


Figure 1. Modeler Hierarchy

1. TWIN is a solid modeling package developed by Timothy Mashburn at Purdue University. It is the basis for his Master's thesis [Mashburn 1987].

3. Synopsis of Modeler Verbs

There are six verbs currently understood by the modeler:

Add The primary verb for creating and modifying a model. Five types of things can be added: features, objects, groups, environments, and applications.

Copy Copies objects (and groups) from one environment to another (possibly the same) environment and environments from one application to another (possibly the same).

Draw Generates line drawing commands to an external display program that is connected to the modeler via a Unix pipe. Three types of things can be drawn: objects, groups, and environments. Scaling and translation can be used.

Get Retrieves information from the model about all types of things.

Read Reads a sequence of FEL sentences stored in a disk file and then presents them to the modeler as if they were typed one by one on the command line.

Transform Applies geometrical transformations to either individual objects, groups, or to all the objects and groups in an environment. Three kinds of transformations are supported: translation, rotation (in X, Y, and Z), and scaling.

4. How Sentences are Processed

A single sentence can do lots of work, because a sentence can contain multiple feature lists. For example, a single *add* sentence can add an object to an environment and then add several features to the object. A *transform* sentence can transform several objects and environments. A single sentence cannot both add and transform an object, because a sentence can have only one verb.

4.1. Order

Each sentence received by the modeler is fully processed before work begins on the next one. The verb is extracted and the indicated action is then performed on each feature list.

The feature lists are processed in order, as they appear in the sentence. In the case of an *add* sentence, this is necessary because the order in which a mixture of positive and negative features are processed affects the result.

In general, attribute-value pairs within feature lists are not processed sequentially, but the *transform* sentence is an exception. Order is significant when rotations about several axes are performed, or when translations and rotations are intermixed in order to rotate an object about a point other than the origin. For these reasons, the transformation

attributes within feature lists of a *transform* sentence are processed in order, as they appear in the feature lists.

4.2. Defaults

It has already been mentioned that there is a default application and, within it, a default environment. They are created when the modeler starts up. Other applications and environments are created by means of *add* sentences.

Every feature list must specify what application, environment, group (if any), object, etc., it is intended to apply to. The modeler remembers the most recently specified application, environment, group, and object; so repeated specification of the same names is not necessary.

5. The Add Verb

The *add* verb is used to add applications, environments, groups, objects and features to the model, but the most common use is to add objects and groups to an environment and features to an object.

5.1. Adding Objects

The following example shows how to add an object to an environment. The TYPE attribute specifies what kind of thing, in this case an object, is to be added and the NAME attribute gives the name which will be used to refer to this thing. The APPLICATION and ENVIRONMENT attributes specify where in the model hierarchy the new object will exist. Currently, a new object can only be created with a rectangular envelope. In a future version of the modeler, arbitrarily shaped envelopes will be allowed.

```
(add
      ...                              ; possibly other feature lists
      ( (type object)                  ; what kind of thing is being added
        (name boomerang)               ; its name
        (application planner)          ; the application name
        (environment part_model)       ; the environment name
        (p_vector (0 0 0))             ; the rectangular envelope
        (w_vector (2.5 0 0))
        (l_vector (0 6.982 0))
        (d_vector (0 0 .25))
      )
      ...
)
```

5.2. Adding Features

We can extend the above example to show how to add a feature to an object. Note that "object", previously the value of the TYPE attribute, is now an attribute whose value is the name of the object to which the feature is being added.

```
(add
      ...                              ; possibly other feature lists
      ( (type thru_slot)               ; what kind of thing is being added
```

```
        (name facemill)              ; its name
        (application planner)        ; the application name
        (environment part_model)     ; the environment name
        (object boomerang)           ; the object name
        (p_vector (0 1.7 0))         ; parameters for a thru_slot
        (w_vector (2.5 0 0))
        (l_vector (0 3.582 0))
        (d_vector (0 0 .1))
      )
      ...
)
```

The above example could have been abbreviated as show below, because the previously specified application, environment and object names will have become the default values for these attributes.

```
(add
      ...
      ( (type thru_slot)             ; what kind of thing is being added
        (name facemill)              ; its name
        (p_vector (0 1.7 0))         ; parameters for a thru_slot
        (w_vector (2.5 0 0))
        (l_vector (0 3.582 0))
        (d_vector (0 0 .1))
      )
      ...
)
```

There are currently eleven features that can be added to an object and they are of two kinds: positive volume and negative volume. Adding a positive volume feature to an object adds volume to the object; adding a negative volume feature removes volume. The positive volume features include block, cylinder, swept_cylinder, and wedge. The negative volume features include face_mill, id_circular_profile, od_circular_profile, thru_slot, thru_hole, swept_radius, and thru_angle.

5.2.1. Block, Face_Mill, and Thru_Slot

A block (or a face_mill or a thru_slot) is defined by four vectors, a position vector and the three vectors that form the corner of the block at that position. The corner vectors determine the size and shape of the block. These four vectors are defined using the attributes P_VECTOR, W_VECTOR, L_VECTOR, and D_VECTOR. The values of these attributes are defined by order triples of real numbers. Here is an example definition of a feature list that describes a thru_slot that is to be created.[2]

2. Recall that a feature list is only meaningful in the context of a sentence that begins with a verb, like "`(add ...)`".

   It is actually unfortunate that new features are described in a manner so closely tied to the underlying geometrical representation. For example, specifying a slot by length, width, depth, and relative position on the part would be much closer to a "feature oriented" description, would be more natural from the point of view of a designer, and would be easier to interface with our feature oriented planner.

```
( (type thru_slot)          ; feature type
  (name slot1)              ; its name
  (p_vector (0 1.7 0))      ; position vector
  (w_vector (2.5 0 0))      ; direction vector
  (l_vector (0 3.582 0))    ; direction vector
  (d_vector (0 0 .1))       ; direction vector
)
```

5.2.2. Blind_Hole, Cylinder, and Thru_Hole

A cylinder (or a thru_hole) is defined by two vectors, a position vector and an axis vector, and a real number, the radius of the cylinder. The position vector defines the center of one end of the (circular) cylinder and the axis vector determines the length and orientation of the cylinder. The vectors are defined using the attributes P_VECTOR and D_VECTOR. The radius is define using the attribute RADIUS. Here is an example definition of a feature list that describes a cylinder.

```
( (type cylinder)           ; feature type
  (name arm_holder)         ; its name
  (p_vector (1.62 .52 1.35)) ; position vector
  (d_vector (0 0.7 0))      ; axis vector
  (radius .31)              ; radius of cylinder
)
```

5.2.3. Swept_Cylinder and Swept_Radius

A swept_cylinder (or a swept_radius) is defined by three vectors and two real numbers. A position vector (attribute P_VECTOR) defines the center of rotation. Relative to the center of rotation, the W_VECTOR attribute gives the center of one end of a cylinder. An axis vector (D_VECTOR) defines the orientation and length of the cylinder and the RADIUS attribute defines its radius as a real number. The attribute ANGLE gives the number of degrees through which the cylinder is to be swept. Here is an example definition of a feature list that describes a swept_radius.

```
( (type swept_radius)       ; feature type
  (name top)                ; its name
  (p_vector (.28 .28 0))    ; center of rotation
  (w_vector (.64 0 0))      ; relative position of cylinder
  (d_vector (0 0 .2825))    ; axis vector
  (angle 360)               ; degrees of sweep
  (radius .5)               ; radius of cylinder
)
```

5.2.4. ID_Circular_Profile and OD_Circular_Profile

A circular profile feature describes a surface that is an arc of a cylinder (see Figure 2, page 58).

```
( (type id_circular_profile) ; feature type
  (name inside_radius)       ; its name
  (p_vector (.28 .28 0))     ; center of rotation
  (r_vector (.64 0 0))       ; far edge of cylinder
  (d_vector (0 0 .2825))     ; axis vector
  (angle 360)                ; degrees of sweep
)
```

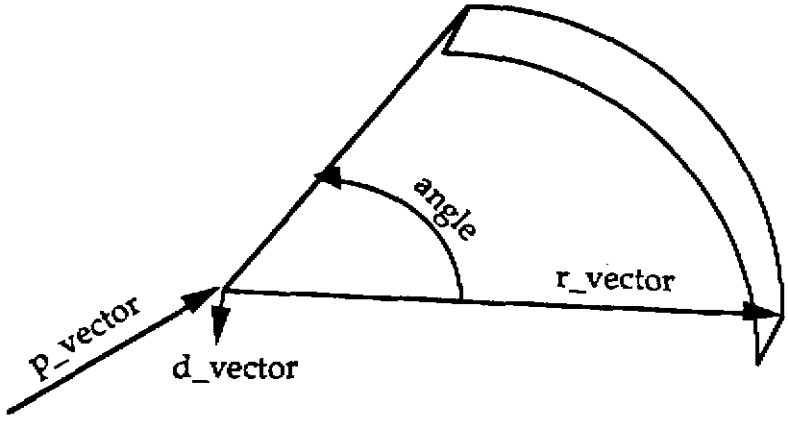


Figure 2. Circular Profile

5.2.5. Wedge and Thru_Angle

A wedge (or a thru_angle) is defined just like a block or thru_slot, but the edge vectors are interpreted differently.

5.2.6. Adding Groups (of objects)

In addition to simple objects, the modeler supports groups of simple objects. Groups can be treated like simple objects, i.e., they can be transformed, copied, drawn, etc. Groups also behave as complex objects whose parts can be manipulated independently. Groups are added to an environment in much the way that simple objects are, but there is no bounding box to be specified.

```
(add
      ...                          ; possibly other feature lists
      ( (type group)               ; what kind of thing is being added
        (name toe_clamp)           ; its name
        (application holding)      ; the application name
        (environment setup1)       ; the environment name
      )
      ...
)
```

5.3. Adding Applications

An application can be explicitly created as shown below. Because all applications are made part of a single predefined *world*, only TYPE and NAME attributes are given.

```
(add
      ...                              ; possibly other feature lists
      ( (type application)             ; what kind of thing is being added
        (name planner)                 ; its name
      )
      ...
)
```

In a future version of the modeler, applications may be created automatically when they are first mentioned.

5.4. Adding Environments

An environment can be explicitly created as shown below. The TYPE and NAME attributes indicate what environment is to be created and the APPLICATION attribute tells where in the model it is to be.

```
(add
      ...                              ; possibly other feature lists
      ( (type environment)             ; what kind of thing is being added
        (name part_model)              ; its name
        (application planner)          ; the application name
      )
      ...
)
```

In a future version of the modeler, environments may be created automatically when they are first mentioned.

6. The Copy Verb

The *copy* verb duplicates an object or an environment, assigns the copy a possibly new name, and places it in a possibly different environment or application. The TYPE, NAME, APPLICATION, and ENVIRONMENT attributes are used to specify what is to be duplicated. The TO_APPLICATION, TO_ENVIRONMENT, and TO_OBJECT attributes are used to specify where the duplicate is to be located and what it should be called.

6.1. Copying Objects

Objects can be copied from one environment to another or duplicated (with a new name) within a single environment. The following example shows how to copy an object to a different environment. It says that an object named "boomerang" in the environment "part_model" of application "planner" is to be copied with the same name to an already existing environment named "part_model2" of the same application.

```
(copy
      ...                              ; possibly other feature lists
      ( (type object)                  ; what kind of thing is being copied
        (name boomerang)               ; its name
        (application planner)          ; source application name
        (environment part_model)       ; source environment name
        (to_application planner)       ; destination application
        (to_environment part_model2)
                                       ; destination environment
        (to_object boomerang)          ; destination object name
      )
      ...
)
```

Duplicating an object in an environment (i.e., when the ENVIRONMENT and TO_ENVIRONMENT attributes have the same value) requires that the duplicate be given a new name (the value of the TO_OBJECT attribute).

6.2. Copying Environments

Environments can be copied from one application to another or duplicated (with a new name) within a single application. The following example shows how to duplicate an environment within an application. It says that an environment named "part_model" in the application "planner" is to be duplicated and that the duplicate is to reside in the same application under the new name "part_model2".

```
(copy
      ...                              ; possibly other feature lists
      ( (type environment)             ; what kind of thing is being copied
        (name part_model)              ; its name
        (application planner)          ; source application name
        (to_application planner)       ; destination application
        (to_environment part_model2)
                                       ; destination environment
      )
      ...
)
```

Duplicating an environment within an application (i.e., when the APPLICATION and TO_APPLICATION attributes have the same value) requires that the duplicate be given a new name (the value of the TO_ENVIRONMENT attribute).

7. The Delete Verb

The *delete* verb deletes an object from an environment or an environment from an application. The TYPE, NAME, APPLICATION, and ENVIRONMENT attributes are used to specify what is to be deleted.

7.1. Deleting Objects

The following example shows how to delete an object from an environment.

```
(delete
      ...                              ; possibly other feature lists
      ( (type object)                  ; what kind of thing is being deleted
        (name boomerang)               ; its name
        (application planner)          ; application name
        (environment part_model)       ; environment name
      )
      ...
)
```

Deleting an object from an environment not only removes the reference to the object from the environment, but also destroys the object structure, its underlying features, associated BREP structures, etc.

7.2. Deleting Environments

The following example shows how to delete an environment from an application.

```
(delete
      ...                              ; possibly other feature lists
      ( (type environment)             ; what kind of thing is being deleted
        (name part_model)              ; its name
        (application planner)          ; application name
      )
      ...
)
```

Deleting an environment from an application not only removes the reference to the environment from the application, but also destroys the environment structure, the objects within the environment, their features, etc.

8. The Draw Verb

The *draw* verb causes a group of display commands to be sent to a separate Unix process via a Unix pipe. The primary application of this verb is the graphic display of an environment of a model on a Sun screen by means of the SunCORE-based MXD program.

8.1. Drawing Objects

The following example shows how to draw an object. The TYPE, NAME, APPLICATION and ENVIRONMENT attributes together specify what is to be drawn. The APPLICATION and ENVIRONMENT attributes may be omitted if the most recently used values are acceptable. Translation and scaling are often needed to bring the desired portion of the model into view. These transformations only apply to the display and have no effect on the model itself.

```
(draw
      ...                              ; possibly other feature lists
      ( (type object)                  ; what kind of thing to draw
        (name boomerang)               ; its name
        (application planner)          ; the application name
        (environment part_model)       ; the environment name
        (translation (0 -.7 0))        ; Y translation
        (scale (.014 .014 .014))       ; X, Y, Z scaling
      )
      ...
)
```

8.2. Drawing Environments

Drawing an environment is similar to drawing an object and the same transformations may be applied. There is, though, a difference between drawing an environment and drawing all the objects in an environment. When an environment is drawn, the viewing surface is automatically cleared before the first object is drawn. When individual objects are drawn, no clearing is done.

```
(draw
      ...                              ; possibly other feature lists
      ( (type environment)             ; what kind of thing to draw
        (name part_model)              ; its name
        (application planner)          ; the application name
        (translation (0 -.7 0))        ; Y translation
        (scale (.014 .014 .014))       ; X, Y, Z scaling
      )
      ...
)
```

8.3. Display Commands

The display commands generated via *draw* consist of a sequence of lines, each containing a single character code followed by some number of parameters.

Close Syntax: `C`
Closes the SunCORE "view_surface" window.

Draw Syntax: `D x y z`
Draws a line from the "current position" to the specified 3D position and updates the "current position".

Line Syntax: `L` x_1 y_1 z_1 x_2 y_2 z_2
Draws a line from the point (x_1 y_1 z_1) to the point (x_2 y_2 z_2).

Move Syntax: `M x y z`
Moves the "current position" to the specified 3D point.

New Page Syntax: `N`
Clears the SunCORE "view_surface" window.

Open Syntax: `O`
Opens and initializes the SunCORE "view_surface" window.

Scale Syntax: `S x_scale y_scale z_scale`
Scales the drawing in each dimension by the specified scale factor.

Translate Syntax: `T x_offset y_offset z_offset`
Translates the drawing in each dimension by the specified amount.

Wait Syntax: `W time`
Wait for the specified number of 1/60th seconds.

X Rotation Syntax: `X x_degrees`
Rotates the drawing around the X axis by the specified number of degrees.

Y Rotation Syntax: `Y y_degrees`
Rotates the drawing around the Y axis by the specified number of degrees.

Z Rotation Syntax: `Z z_degrees`
Rotates the drawing around the Z axis by the specified number of degrees.

8.4. Alternate Display Programs

When the modeler (mx) is executed, the name of an (optional) executable "display" program may be provided as an argument. For example, if "mxd" is the name of such a display program, the modeler could be executed with the following command.

```
% mx mxd
```

This display program is executed as a "child" process under the modeler and is connected to the modeler via a "pipe" so that the display process can receive commands from the modeler via the display process' standard input. The commands that it may receive are described in the previous section.

A display program (mxd) that produces perspective drawings on the Sun (using the SunCORE library) is currently available. Simple modifications of this program could be used to generate PostScript or types of output from the modeler.

9. The Get Verb

The *get* verb is used to retrieve information from the modeler. The format for retrieving applications, environments, objects, and features is illustrated in the example below. Each feature list specifies the type, name, and appropriate context of the thing to be retrieved. The amount and kind of information supplied by the modeler depends on the type of thing requested.

9.1. Application Request

When you want to know what environments are contained in an application, you send the modeler a *get* message with TYPE "application". You specify the name of the application you want with the NAME attribute. Here's how to get a list of environments in the "planner" application:

```
(get
      ...
      ( (type application)          ; what kind of thing to get
        (name planner)              ; its name
      )
      ...
)

(got
      ...
      ( (type application)          ; type of thing retrieved
        (name planner)              ; its name
        (world world)               ; the world name
        (environment (part_model)) ; list of environments in this
                                    ; application
      )
      ...
)
```

The reply repeats the TYPE and NAME attributes, tells you the name of the "world" containing the application, and gives the list of environments as the value of the ENVIRONMENT attribute. You can use this information to retrieve information about each of the environments.

9.2. Environment Request

When you want to know what objects are contained in an environment, you send the modeler a *get* message with TYPE "environment". You specify the name of the environment you want with the NAME attribute. You specify the application containing this environment with the APPLICATION attribute. Here's how to get a list of objects in the "part_model" environment:

```
(get
      ...
      ( (type environment)          ; what kind of thing to get
        (name part_model)           ; its name
        (application planner)       ; the application name
      )
      ...
)

(got
      ...
      ( (type environment)          ; type of thing retrieved
        (name part_model)           ; its name
        (world world)               ; the world name
```

```
        (application planner)      ; the application name
        (object (boomerang))       ; list of objects in this environment
      )
      ...
)
```

The reply repeats the TYPE, NAME, and APPLICATION attributes, tells you the name of the "world" containing the application, and gives the list of objects as the value of the OBJECT attribute. You can use this information to retrieve information about each of the objects.

9.3. Object Request

When you want to know what features are contained in an object, you send the modeler a *get* message with TYPE *"object"*. You specify the name of the object you want with the NAME attribute. You specify the application and environment containing this object with the APPLICATION and ENVIRONMENT attributes. Here's how to get a list of features in the "boomerang" object:

```
(get
      ...
      ( (type object)              ; what kind of thing to get
        (name boomerang)           ; its name
        (application planner)      ; the application name
        (environment part_model)   ; the environment name
      )
      ...
)

(got
      ...
      ( (type object)              ; type of thing retrieved
        (name boomerang)           ; its name
        (world world)              ; the world name
        (application planner)      ; the application name
        (environment part_model)   ; the environment name
        (feature (envelope hole1)) ; list of features of this object
      )
      ...
)
```

The reply repeats the TYPE, NAME, APPLICATION and ENVIRONMENT attributes, tells you the name of the "world" containing the application, and gives the list of features as the value of the FEATURE attribute. You can use this information to retrieve information about each of the features.

9.4. Feature Request

When you want to know about a particular feature in an object, you send the modeler a *get* message with TYPE *"feature"*. You specify the name of the name you want with the NAME attribute. You specify the application, environment, and object containing this feature with the APPLICATION, ENVIRONMENT, OBJECT attributes. Here's how to get

information about a feature named "hole1":

```
(get
      ...
      ( (type feature)              ; what kind of thing to get
        (name hole1)                ; its name
        (application planner)       ; the application name
        (environment part_model)    ; the environment name
        (object boomerang)          ; the object name
      )
      ...
)

(got
      ...
      ( (type thru_hole)            ; type of thing retrieved
        (name hole1)                ; its name
        (world world)               ; the world name
        (application planner)       ; the application name
        (environment part_model)    ; the environment name
        (object boomerang)          ; the object name

      )
      ...
)
```

The reply repeats the NAME, APPLICATION, ENVIRONMENT and OBJECT attributes, tells you the name of the "world" containing the application, includes a TYPE attribute with the specific type of the feature, and gives other attribute/value pairs appropriate for that feature.

The following sections describe the special attribute/value pairs for each of the features supported by the modeler.

9.4.1. BREP

The BREP pseudo-feature gives a boundary representation of an object and is included in the modeler for use by the Holding Expert, which employs preexisting algorithms that require this kind of non-feature-oriented input. Three attributes are provided in addition to the standard ones that accompany all feature descriptions.

```
(got
      ...
      (...
        (vertex ((x1 y1 z1)...(xn yn zn)))
        (loop ((v11...v1m)...(vp1...vpn)))
        (face ((l11...l1m)...(lp1...lpn)))
      )
      ...
)
```

The VERTEX attribute gives a list of all vertices of the object. A vertex is a triple of real numbers giving the x-y-z coordinates of a point. The LOOP attribute gives a list of all loops of the object. A loop is a sequence of vertices that describe a connected sequence of edges on a face. Each vertex in a loop is described by an integer index into the VERTEX list. The FACE attribute gives a list of all the faces of the object. A face is described by a list of loops, the first of which is its outer boundary; subsequent loops on a face describe holes in the face. Each loop is described by an integer index into the LOOP list.

9.4.2. Envelope

The ENVELOPE feature describes a rectangular bounding box around an object. Three attributes are given in addition to the standard ones that accompany all feature descriptions: FINISH, DISTANCE, and NORMAL.

```
(got
      ...
      ( (finish ((side1 f1)...(side6 f6)))
        (distance ((s1 s2 d1) (s3 s4 d2) (s5 s6 d3)))
        (normal ((side1 (x1 y1 z1))...(side6 (x6 y6 z6))))
      )
      ...
)
```

The FINISH attribute specifies the surface finish of the sides of the stock. It is given as a list of pairs, each of which is composed of the name of a side and the name of the finish property for that side. The DISTANCE attribute specifies the distance between parallel sides of the envelope. It is given as a list of three triples, each of which has two side names and the real numbered distance between those sides. The NORMAL attribute unit face normals of the sides of the stock. They are given as a list of pairs, each of which is composed of the name of a side and a vector, represented as a three element list.

9.4.3. Thru_Hole

The THRU_HOLE feature describes a hole that opens on two opposite sides of the envelope of an object. Five attributes are provided in addition to the standard ones that accompany all feature descriptions.

```
(got
      ...
      ( (position ((x1 y1 z1) (x2 y2 z2)))
                                  ; position on true faces corresponding
                                  ; to the "opens_on" sides
        (depth d)                 ; depth of the hole
        (radius r)                ; radius of the hole
        (opens_on (side1 side2))  ; envelope sides
        (distance ((side3 dist1) (side4 dist2)))
                                  ; distance from orthogonal sides
      )
```

```
        ...
    )
```

The POSITION attribute gives the position of the hole on each of the surfaces that it opens onto, in the order that corresponds with the envelope sides given by the OPENS_ON attribute. The DEPTH attribute gives the depth of the hole as a real number. The RADIUS attribute gives the radius of the hole as a real number. The OPENS_ON attribute specifies the two opposite sides of the envelope associated with the hole. The DISTANCE attribute tells the distance of the center of the hole from two orthogonal sides of the envelope of the object.[3]

9.4.4. Blind_Hole

The BLIND_HOLE feature describes a hole that opens on only one side of the envelope of an object. Five attributes are provided in addition to the standard ones that accompany all feature descriptions.

```
(got
      ...
      ( (position (x1 y1 z1))        ; position on "opens_on" side
        (depth d)                    ; depth of the hole
        (radius r)                   ; radius of the hole
        (opens_on (side1))           ; envelope side the hole is "on"
        (distance ((side3 dist1) (side4 dist2)))
                                     ; distance from orthogonal sides
      )
      ...
)
```

The POSITION attribute gives the position of the hole on the surface that it opens onto; this surface corresponds to, but may not be identical with, the envelope side supplied by the OPENS_ON attribute. The DEPTH attribute gives the depth of the hole as a real number. The RADIUS attribute gives the radius of the hole as a real number. The OPENS_ON attribute specifies the side of the envelope associated with the hole. The DISTANCE attribute tells the distance of the center of the hole from two orthogonal sides of the envelope of the object.[4]

9.4.5. Channel (Thru_Slot)

The CHANNEL feature describes a slot that opens on three sides of the envelope of an object. Four attributes are provided in addition to the standard ones that accompany all feature descriptions: OPENS_ON, WIDTH, DEPTH, and DISTANCE.

3. For the cutting expert, we also supply the original vectors (possibly transformed) used to define this feature. These vectors are the values of the attributes P_VECTOR and D_VECTOR.
4. See footnote 3.

```
(got
      ...
      ( (depth d)                    ; depth of the channel
        (width w)                    ; width of the channel
        (opens_on (s1 s2 s3))        ; envelope sides cut by the channel
        (distance (side dist))       ; distance from one side
      )
      ...
)
```

The POSITION attribute gives the position of the hole on the surface that it opens onto; this surface corresponds to, but may not be identical with, the envelope side supplied by the OPENS_ON attribute. The DEPTH attribute gives the depth of the channel as a real number. The WIDTH attribute gives the width of the channel as a real number. The OPENS_ON attribute specifies the three sides of the envelope associated with the channel. The DISTANCE attribute tells the distance from one side of the envelope of the object to the nearest side of the channel.[5]

10. The Read Verb

The *read* verb[6] allows input of FEL sentences from one or more files, which are specified by the feature lists. This is especially useful for loading models of the relatively unchanging machining environment. Each such feature list must contain the attribute-value pair "`(type file)`" and a pair with attribute NAME whose value is the name of a file. The following sentence would cause two files to be read and the sentences in them to be processed before input from any other source, such as the network, is processed.

```
(read
      ...
      ( (type file) (name "model.fel") )
      ( (type file) (name "model-transforms.fel") )
      ...
)
```

11. The Transform Verb

The *transform* verb is used to apply geometric transformations to one or more objects. Translation, rotation, and scaling are supported. Unlike transformations applied with the *draw* verb, these actually alter the coordinates of the vertices of the objects. Both environments and individual objects may be transformed. Transforming an environment has the same result as individually applying the same transformation to all the objects in that environment.

What is to be transformed (either object or environment) and what transformation (a combination of translation, rotation, and scaling) is to be applied are specified by one or more feature lists. If an individual object is to be transformed, then the list should con-

5. See footnote 3.

6. This is not, strictly speaking, a "modeler verb." Sentences containing this verb are processed by the Network task in the Generic Expert and are never passed on to the application-specific Expert task. So, the Model task in the MX application never sees such sentences.

tain the attribute-value pair "(type object)" and a pair with attribute NAME whose value is the name of the object. If an environment is to be transformed, the TYPE attribute should have the value "environment". Here is a typical *transform* sentence.

```
(transform
      ( (type environment) (name env1)
        (translation (1. 1. 0.)) (scale (2. 2. 2.))
      )
      ( (type object) (name obj2)
        (translation (-1. -1. 0.))
        (z_rotation 30.)
        (translation (1. 1. 0.))
      )
)
```

11.1. Translation

The TRANSLATION attribute takes a vector (i.e., a list with exactly three real numeric elements) as its value. The elements of the vector specify the translation in X, Y, and Z, respectively. For example, the following sentence adds 1 to the X components and -2 to the Y components of all vertices of all objects in the environment named "env1".

```
(transform
      ( (type environment) (name env1) (translation (1. -2. 0)) )
)
```

11.2. Rotation

There are three rotation attributes: X_ROTATION, Y_ROTATION, and Z_ROTATION. Each takes a real numeric value which specifies the number of degrees of rotation about the X, Y, and Z axes, respectively. Positive rotations are clockwise from the point of view of a positive axis, looking towards the origin. The following example rotates all the objects in the specified environment by 90 degrees about the X axis.

```
(transform
      ( (type environment) (name env1)
        (x_rotation 90)
      )
)
```

11.3. Scaling

The SCALE attribute takes a vector (i.e., a list with exactly three numeric elements) as its value. The elements of the vector specify the scaling in X, Y, and Z, respectively. The following example shrinks all the objects in the specified environment to one fourth their previous size.

```
(transform
      ( (type environment) (name env1)
        (scale (0.25 0.25 0.25))
      )
)
```

12. Bibliography

Mashburn 1987 Mashburn, Timothy Allen, *A Polygonal Solid Modeling Package*, Masters Thesis, Purdue University, 1987.

Center for
Integrated Manufacturing Decision Systems

FEL Interface for Communicating with the Holding Expert

Jeff Baird

March, 1990

Abstract:

The Holding Expert selects and positions fixtures for each setup planned by the IMW (Intelligent Machining Workstation) system. This document describes the FEL sentences accepted and generated by the Holding Expert (HX).

A description of the Holding Expert's communication with other IMW subsystems is given. The bulk of the document is a detailed description of the semantics of every VERB, feature list TYPE, and ATTRIBUTE handled by the current prototype version of the Holding Expert.

Copyright © 1990 Carnegie Mellon University

Contact:

David Bourne
CIMDS
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-8810

1. Introduction

This document describes the FEL interface to the IMW Holding Expert. The interface is a mapping of the program inputs and outputs into FEL sentences. The holding expert was developed by Kyoung Kim and is described in his doctoral thesis (forthcoming). This document provides an explanation of the input requirements and provides a complete reference to the messages the program sends and receives.

The IMW is composed of several different expert systems, each with its own area of expertise in different machining tasks. The primary systems are for planning, holding, cutting, sensing, modeling and execution/control. In order to achieve the goal of machining a part, these different expert systems must exchange information and agree upon constraints.

The goal of the holding expert system is to provide a fixturing plan for a part to the execution/control system that will hold the part securely during machining and allow good access to the part for the cutting and sensing systems. The fixturing plan is a list of fixtures from a library, a sequence of NC programs for the IMW novel tooling to place the fixtures, and commands to update the modeler with the changes in the environment that the plan causes (e.g., so other systems querying the modeler see that fixtures have been placed in the environment.)

To generate the fixturing plan, the holding expert needs several pieces of data. It needs to know the shape of the part, the location and orientation of the part on the machine tool bed, and the access paths that the cutting system would like to use.

The shape or geometry of the part is maintained by the modeler. A simple bounding box or part envelope approach is not sufficient enough for the holding expert. Toe clamps cannot be placed on holes in the part and vises cannot use curved surfaces. A complete description of the part is required in order to position fixtures in a rigid, error-free manner. The modeler maintains a complete boundary representation (BREP) of the part that can be used by the holding expert to locate good surfaces for fixturing.

The location and orientation of the part are determined by the planner. The planner orders the making of features (holes, shoulders, slots, ...) which determines the location and orientation of the part (e.g., on a 3 axis milling machine, the side to have a hole drilled in it must be facing "up".) This ordering process in the planner strongly impacts the flow of data among the expert systems. Since the planner orders features, several features may be machined within one fixturing setup. Effectively, the entire system is driven on a setup by setup basis.

The primary task for the holding expert is to plan the fixturing for each setup generated by the planner. The planner also can affect the part location and orientation by selecting special fixtures to aid in making the part. The part might be on top of a subplate because it is thin. The part might have an angled surface that must be made by mounting the part to a sine table. The use of these methods must be conveyed to the holding expert. The planner also knows which part surface is best to treat as a reference surface. A reference surface is used to accurately locate the part (e.g., a milled surface is usually better than a saw cut one. So the milled surface would be placed against the locating pins.) The planner's selection of a reference surface must also be sent to the holding expert.

The cutting expert produces tool paths to produce the features in each setup. Obviously the

fixtures cannot interfere with the cutting paths. The cutting expert also selects tools and feeds and speeds. The tool diameter and the horsepower along a cutting trajectory impose force vectors upon the part that the holding expert must negate in order to hold the part stable. Each tool path and parameters must be given to the holding expert for each setup.

Although the above paragraphs describe the logical flow of information in and out of the holding expert, the actual flow in the IMW systems is a bit different. The control/execution system, sometimes referred to as the plan manager, is collecting output from each system for execution. It has the planner part location information and the cutting expert tool paths and the "path" in the modeler of the part geometry. (Since machining a part typically involves multiple setups, the modeler has different models for each setup. The modeler maintains a named hierarchy of models and the control/execution system knows which names correspond to each setup. These names form a path that can be given to the holding expert to retrieve the part boundary representation (BREP) for the part in the current setup.)

The control/execution system can send a FEL message to the holding expert with the above information, then the holding expert need only query the modeler (using the path from the control/execution system) to get the BREP. Then the holding expert will have all the data needed to produce a plan fixturing the current setup. This basic cycle of "control tells holding to plan, holding queries modeler and then produces a plan" is repeated for each setup that the planner generates.

If the holding expert can generate a fixturing plan, it will return the plan to the control/execution system. Although the plan contains an ordered list of fixtures, the important output is actually an NC program. The name of a file containing NC programs to physically place the fixtures at the appropriate locations is returned. Also the geometric models of the fixtures to be used are moved to the appropriate place (in the environment that the planner named (path)) from a library of fixture models in the modeler.

The sections that follow are a very detailed list of the FEL sentences and their components that implement the interface described above. They are intended to be used as a reference for implementation. They do not explain functions and terms from other systems. Readers should be familiar with FEL, the generic expert, and the modeling expert.

2. Flow of Sentences

The basic flow of sentences in and out of the holding expert (called HX) is as follows:

HX receives a PLAN a SETUP message. PLAN messages are typically from the plan manager (PL). The PLAN sentence contains one type SETUP featurelist and zero or more type TOOLPATH featurelists. The SETUP describes the location of the part model in the modeler, its orientation, reference sides, and the planner's fixturing methods. Each TOOLPATH describes the cutter size, start and stop points of the path, feed rate, horsepower, and the kind of cut.

Using the setup information HX then sends a GET BREP message to the modeler MX. The BREP is the boundary representation of the part. The edges and surfaces of the part are defined in the BREP.

When HX receives GOT BREP from the modeler, it converts the BREP's VERTEX, LOOP, FACE, and NORMAL data plus the TOOLPATHs into structures usable by the main fixture planning routine and calls it. The fixture planning routine returns a list of fixtures and the name of a file containing NC code to position the fixtures.

HX then sends a COPY sentence and a TRANSFORM sentence to MX. The COPY message copies each fixture from a library to the current part model. The TRANSFORM message moves each fixture into the correct position and orientation in the part model.

When HX receives the COPIED message and then the TRANSFORMED message back from MX, it sends a PLANNED BUILD_SETUP message back to the originator of the PLAN message (usually PL). (It should really send back a PLAN SETUP message, but due to some restrictions in the plan manager software, it uses BUILD_SETUP.) The PLANNED message contains a list of the fixture names used and the name of the NC file.

Now HX is ready to receive the next PLAN SETUP message.

If any negated verbs (NOTGOTTEN, NOTCOPIED, or NOTTRANSFORMED) are received from MX or the fixture planning routine cannot give a list of fixtures to hold the part, a NOTPLANNED message is sent back to the originator of the PLAN SETUP message.

In the current version of HX, version 0.6, only one PLAN message at a time will be handled. PLAN requests sent while planning is already in progress will print an error message on the terminal and will then be IGNORED! (no NOTPLANNED is sent back).

3. Verbs

The holding expert accepts a very limited number of FEL verbs. The PLAN verb is the only verb used to command the holding expert (sent from another expert or typed in to the generic expert terminal interface.) All other verbs are generated or are received as replies from the modeling expert.

PLAN	The PLAN verb is the primary input that drives the holding expert. It creates a plan for fixturing a setup using the information in its featurelists and will query the modeler for part geometry information. PLAN has two featurelists (TYPES) that it uses. The first is one type SETUP featurelist. The second is zero or more type TOOL_PATH featurelists. See the Attributes description for further details.
GOT	The GOT verb is the reply from the modeler containing the part BREP. Only the featurelist type BREP is allowed here.
NOTGOTTEN	The NOTGOTTEN verb is the reply from the modeler that the part BREP was not retrieve do to some error (See the Errors section.) All featurelists of this verb are ignored.
COPIED	The COPIED verb is the reply from the modeler that the fixtures have been copied from the library into the current part environment. All featurelists of this verb are ignored.
NOTCOPIED	The NOTCOPIED verb is the reply from the modeler that there was an error

copying the fixtures from the library into the current part environment (See the Error section.) All featurelists of this verb are ignored.

TRANSFORMED The TRANSFORMED verb is the reply from the modeler that the fixtures have been transformed to the correct location in the current part environment. All featurelists of this verb are ignored.

NOTTRANSFORMED
The NOTTRANSFORMED verb is the reply from the modeler that there was an error transforming the fixtures to the correct location in the current part environment (See the Error section.) All featurelist of this verb are ignored.

There are three other verbs that the holding expert accepts: ADD, DELETE, and INSPECT. These were used in debugging during development and will not be documented here because they should not be used!

The holding expert generated a very limited number of fel verbs. Only the PLANNED or NOTPLANNED verbs are sent back to the originator of the plan request. The other verbs are generated as requests to the modeling expert to retrieve or update feature information.

PLANNED The primary output of the holding expert is return in the PLANNED verb to the originator of the PLAN request. It has one featurelist type BUILD_SETUP (Due to some limitations in the plan manager software, it returns BUILD_SETUP instead of SETUP.)

NOTPLANNED The NOTPLANNED verb is returned to the originator of the PLAN request whenever a plan cannot be made. This could be from an error in the input (PLAN verb) or from an incorrect modeler environment or from lack of fixtures for the current part configuration/size. This verb has one featurelist type BUILD_SETUP (see BUILD_SETUP in the PLANNED verb above) which will have an error message in the ERRORS attribute.

GET The GET verb is sent to the modeler from the holding expert to retrieve part geometry information. It has one featurelist type BREP which contains the name of the object, environment, and application of the part.

COPY The COPY verb is sent to the modeler from the holding expert to copy the fixtures used in the plan from the fixture library to the current part environment. It contains one type of featurelist type OBJECT which will be repeated for each different fixture used in the plan.

TRANSFORM The TRANSFORM verb is sent to the modeler from the holding expert to transform the fixtures (placed by the COPY verb, see above) into the correct position in the current part environment. It contains one type of featurelist type OBJECT which will be repeated for each different fixture used in the plan. If the COPY verb could also translate each object copied, this verb would be unnecessary.

Verbs other than those listed above cause the holding expert to print out a error message and to ignore the rest of the sentence. A negated verb is not returned for unsupported verbs.

4. FEL Types

Only a few types (FeatureList types) are accepted by the holding expert. The ones actually used in input sentences are SETUP and TOOLPATH. Other message types are primarily replies to queries. The input types are

SETUP (in PLAN) This feature list contains all the planner specific information required by HX to make a fixturing plan. It also contains the location in the modeler (MX) that the part boundary representation can be retrieved from.

TOOLPATH (in PLAN)
This feature list contains the a list of tool paths generated by the cutting expert (CX) to make the features used in the current setup.

BREP (in GOT) This feature list contains the part boundary representation returned from the modeler (MX).

The main Type (FeatureList type) generated by the holding expert is BUILD_SETUP. It contains the fixturing plan information. Other types generated by the holding expert are modeler operations. The types generated by the holding expert are

OBJECT (in GET and COPY and TRANSFORM)
This feature list names the part or a fixture to be retrieved/manipulated by the modeler (MX).

BUILD_SETUP (in PLANNED)
This feature list contains the finished fixturing plan to be returned to the plan manager.

The type HOLDING_OP is accepted and ignored. All other types generate an error message and are ignored. Types returned from the modeling experts COPIED/NOTCOPIED and TRANSFORMED/NOTTRANSFORMED verbs are completely ignored. Unless mentioned as optional, all attributes listed under each type are required.

5. Attributes for SETUP in PLAN

The attributes for PLAN SETUP messages contain the information from the planner that the holding expert needs to generate a fixturing plan. (The holding expert also needs information from the modeler and the cutting expert to complete the plan.)

NAME The value of this attribute is a string which is used to identify the setup. It is returned in the PLANNED and NOTPLANNED messages.

APPLICATION The value of this attribute is a string that names the application in the modeler that has the part description.

ENVIRONMENT The value of this attribute is a string that names the environment in the modeler that has the part description.

PART The value of this attribute is a string that names the object in the modeler that has the part description. The obsolete attribute OBJECT if PART is not present.

FINISHED_PART The value of this attribute is a string. Currently not used except that it is required to be returned to PL.

SETUP_NO The value of this attribute is an integer that is unique for each setup. Used to generate unique NC_FILENAMES.

X_ROTATION The value of this attribute is an real in units of degrees that is the inverse of the last X axis rotation applied to the part. See the ANGLE attribute below.

Y_ROTATION The value of this attribute is an real in units of degrees that is the inverse of the last Y axis rotation applied to the part. See the ANGLE attribute below.

Z_ROTATION The value of this attribute is an real in units of degrees that is the inverse of the last Z axis rotation applied to the part.

TRANSLATION The value of this attribute is a vector of 3 reals that were the last translations applied to the part.

METHOD The value of this attribute is a symbol that is the name of the suggested fixturing method from the OPS5 planner. Must be one of SUBPLATE, VISE, SINE_TABLE, or ANGLE_PLATE. It is required, but currently not used by main fixture planning routine.

SUBPLATE_DEPTH
The value of this attribute is an real that is the height (along the Z axis) of the subplate used in fixturing. If METHOD SUBPLATE is not used it should have the value 0.0. The obsolete attribute DEPTH is checked if SUBPLATE_DEPTH is not present.

ANGLE The value of this attribute is an real in units degrees that is the angle of the sine table that the part is on. If METHOD SINE_TABLE is not used it should be the value 0.0. When using ANGLE, the POSITION attribute should be set to the axis of which this angle of rotation is applied too. WARNING! It is unclear if it is still necessary to have one of the X_ROTATION or Y_ROTATION attributes set to the inverse (negation) of angle so that the axis can be determined.

POSITION The value of this attribute is a symbol that is the name of the axis of rotation of the ANGLE attribute. Must be one of X, or Y.

MAJOR_NORMAL
The value of this attribute is a vector of 3 reals that is the unit normal vector of the modeler envelope face that corresponds to the OPS5 planner's major reference face in the current part coordinate system in the APPLICATION, ENVIRONMENT, and PART(=OBJECT).

MINOR_NORMAL
The value of this attribute is a vector of 3 reals that is the unit normal vector of the modeler envelope face that corresponds to the OPS5 planner's minor reference face in the current part coordinate system in the APPLICATION, ENVIRONMENT, and PART(=OBJECT).

6. Attributes for TOOLPATH in PLAN

Use of the TOOLPATH featurelist is more complex than just a list of attribute values. It has several different forms and ordering restrictions. The toolpaths are a the actual cutter paths to make the part (specified in the part model coordinates.) Each request to PLAN a setup has zero or more type TOOLPATH featurelists. If none are provided, then the holding expert will make

an extremely conservative fixturing plan assuming that no access to the part is needed. There are four different kinds of toolpaths: RAPID, LINEAR, TAPPING, CIRCULAR. Exactly one of these name must be present in each TOOLPATH featurelist. In every PLAN message, the first TOOLPATH featurelist should be a RAPID attribute. The last featurelist should also be a RAPID, but in the current version of HX this is not checked for and can be omitted. The RAPID attribute commands the machine tool to move to a position without cutting. It is used purely for positioning the tool to the place to start cutting. Each successive cut starts at the place where the last cut left off. This is why the first TOOLPATH featurelist of each PLAN is a RAPID, it sets the "last" cutting position before the first cut. The last TOOLPATH featurelist should be a rapid to home the tool to a safe position. After the initial RAPID, any sequence of LINEAR, CIRCULAR, TAPPING, or RAPID is permitted. The LINEAR, CIRCULAR, and TAPPING attributes specify paths that are actually cutting metal. Each one of these attributes start from the position the previous TOOLPATH featurelist left off at. Since RAPID does not cut metal, the HORSEPOWER, SFM, and DIAMETER attributes are ignored when RAPID is present. The HORSEPOWER, SFM, and DIAMETER attributes are required for the attributes that cut: LINEAR, CIRCULAR, and TAPPING. The CIRCULAR attribute also requires center point for the circular path to be specified in the CENTER attribute. See the Sample Sentences section for some examples of using TOOLPATHS.

NAME	The value of this attribute is a string which is used to identify the tool being used in each cut. Currently, this attribute is not referenced.
RAPID	The value of this attribute is a vector of 3 reals that set the starting position for the next cutter path.
LINEAR	The value of this attribute is a vector of 3 reals that forms the end of a linear cutting path. The last cutting path position is used as the starting point.
TAPPING	The value of this attribute is a vector of 3 reals that forms the end of a tapping cutting path. Currently, the holding expert treats this identically to the LINEAR attribute. The last cutting path position is used as the starting point.
CIRCULAR	The value of this attribute is a vector of 3 reals that forms the end of a circular cutting path. The last cutting path position is used as the starting point. Using this attribute requires that a CENTER point for the circular path appear in this TOOLPATH.
HORSEPOWER	The value of this attribute is a real that is the maximum horsepower used during this TOOLPATH.
SFM	The value of this attribute is a real that is the feed rate of the tool table used during this TOOLPATH. If SFM is not present, the obsolete attribute SPEED is check for.
DIAMETER	The value of this attribute is a real that is the diameter of the cutter used during this TOOLPATH.
CENTER	The value of this attribute is a vector of 3 reals that forms the center point of a circular cutting path. This attribute must be present if the CIRCULAR attribute is used.

The are several obsolete attribute for TOOLPATHS that are recognized: RADIUS, P_VECTOR, and D_VECTOR. They will not be documented because they should not be used.

7. Attributes for BREP in GET

The attributes in the GET BREP (boundary representation) message are used to query the modeler to get BREP for the current setup. These attributes are generated automatically from attribute values in the PLAN SETUP message.

NAME — The value of this attribute is a symbol which is the name of current part object in the modeler. The value used is the value of the PART attribute from the PLAN verb. This value is the same as the value of the OBJECT attribute.

APPLICATION — The value of this attribute is a symbol which is the name of current part application in the modeler. The value used is the value of the APPLICATION attribute from the PLAN verb.

ENVIRONMENT — The value of this attribute is a symbol which is the name of current part environment in the modeler. The value used is the value of the ENVIRONMENT attribute from the PLAN verb.

OBJECT — The value of this attribute is a symbol which is the name of current part object in the modeler. The value used is the value of the PART attribute from the PLAN verb. This value is the same as the value of the NAME attribute.

8. Attributes for BREP in GOT

The attribute values in the GOT BREP (boundary representation) message are the values returned by the modeler from a GET BREP query. They contain the part description that the holding expert uses to locate good surfaces for fixturing.

NAME — The value of this attribute is a symbol which is the name of current part object in the modeler. The value received is the value of the NAME attribute from the GET verb. This value is the same as the value of the OBJECT attribute.

APPLICATION — The value of this attribute is a symbol which is the name of current part application in the modeler. The value received is the value of the APPLICATION attribute from the GET verb.

ENVIRONMENT — The value of this attribute is a symbol which is the name of current part environment in the modeler. The value received is the value of the ENVIRONMENT attribute from the GET verb.

OBJECT — The value of this attribute is a symbol which is the name of current part object in the modeler. The value received is the value of the PART attribute from the GET verb. This value is the same as the value of the NAME attribute.

WORLD — The value of this attribute is a symbol which is the name of current part world in the modeler. This value is not used.

VERTEX — The value of this attribute is a list of vectors of 3 reals of the vertices of the part named by APPLICATION, ENVIRONMENT, and OBJECT. It is converted into an array for use by the main fixture planning routine.

LOOP — The value of this attribute is a list of vectors of 3 reals of the TWIN loops of the part named by APPLICATION, ENVIRONMENT, and OBJECT. It is converted into an arrays HOLES and FACES for use by the main fixture planning routine.

FACE — The value of this attribute is a list of vectors of 3 reals of the TWIN faces of the part named by APPLICATION, ENVIRONMENT, and OBJECT. It is used to guide which LOOPS are converted into HOLES and into FACES for use by the main fixture planning routine.

NORMAL — The value of this attribute is a list of vectors of 3 reals of the TWIN face normals of the part named by APPLICATION, ENVIRONMENT, and OBJECT. It is included in the array of FACES for use by the main fixture planning routine.

9. Attributes for BUILD_SETUP in PLANNED

The BUILD_SETUP featurelist in PLANNED is the primary output of the holding expert. It returns the part rotations and translations and the list of fixtures to use and the name of an NC code file that has commands for placing the fixtures. Due to some limitations in plan manager software, it also returns all the attributes sent down in the PLAN SETUP message. Only X_ROTATION, Y_ROTATION, Z_ROTATION, TRANSLATION, NC_FILENAME, FIXTURES, and MODE contain useful, new information. All other attributes of this sentence are copied from the original PLAN sentence.

NAME — The value of this attribute is a symbol which is the name of the setup that was planned. Its value is the value of the attribute NAME for SETUP in PLAN.

X_ROTATION — The value of this attribute is an real in units of degrees that is the X axis rotation to be applied to the part for fixturing this setup. In the current version of HX, this value is not used (always 0.0.)

Y_ROTATION — The value of this attribute is an real in units of degrees that is the Y axis rotation to be applied to the part for fixturing this setup. In the current version of HX, this value is not used (always 0.0.)

Z_ROTATION — The value of this attribute is an real in units of degrees that is the Z axis rotation to be applied to the part for fixturing this setup. In the current version of HX, this value is the only rotation actually used, (X and Y are not.)

TRANSLATION — The value of this attribute is a vector of 3 reals that are translations to be applied to the part for fixturing this setup. In the current version of HX, this value is not used (always (0.0 0.0 0.).)

NC_FILENAME — The value of this attribute is a string that is the name of a file that has the NC code to place fixtures for the current setup. This file is generated for each setup planned and thus must have a unique name. The name is of the form hx_macro_Number, where Number is the value of the SETUP_NO attribute from SETUP in PLAN. The file is generated in the current working directory of the HX process, so an absolute path (filename start with a /) is provided.

FIXTURES The value of this attribute is a list of symbols that the main fixture planning routine choose to fixture the part for the current setup. These each of these symbols have the form Name_Number where Name is one of LOCATOR, TOE_CLAMP, FIXED_VISE_JAW, MOVEABLE_VISE_JAW, PARALLEL_BAR, RISER, BOLT, SUBPLATE. Each type of fixture is uniquely identified by appending a number to its name. If the fixture library has three locators, they would be referenced by LOCATOR_1, LOCATOR_2, and LOCATOR_3.

MODE The value of this attribute is a symbol that the says whether the fixturing operations are done automatically or manually. Its value is one of MANUAL or AUTOMATIC. Due to limitations in plan manager software this is returned in BUILD_SETUP always as MANUAL. When the limitations are removed, the per fixture mode generated in the NC file will be used instead.

SETUP_NO The value of this attribute is an integer that is unique for each setup. It is Used to generate unique NC_FILENAMES. It is the value of the corresponding attribute from SETUP in PLAN.

APPLICATION The value of this attribute is a string that names the application in the modeler that has the part description. It is the value of the corresponding attribute from SETUP in PLAN.

ENVIRONMENT The value of this attribute is a string that names the environment in the modeler that has the part description. It is the value of the corresponding attribute from SETUP in PLAN.

PART The value of this attribute is a string that names the object in the modeler that has the part description. It is the value of the corresponding attribute from SETUP in PLAN.

FINISHED_PART The value of this attribute is a string. It is currently not used. It is the value of the corresponding attribute from SETUP in PLAN.

ANGLE The value of this attribute is an real in units degrees that is the angle of the sine table that the part is on. See SETUP in PLAN for more information about its use. It is the value of the corresponding attribute from SETUP in PLAN.

SUBPLATE_DEPTH
The value of this attribute is an real that is the height (along the Z axis) of the subplate used in fixturing. It is the value of the corresponding attribute from SETUP in PLAN.

POSITION The value of this attribute is a symbol that is the name of the axis of rotation of the ANGLE attribute. Must be one of X, or Y. It is the value of the corresponding attribute from SETUP in PLAN.

METHOD The value of this attribute is a symbol that is the name of the suggested fixturing method from the OPS5 planner. Must be one of SUBPLATE, VISE, SINE_TABLE, or ANGLE_PLATE. It is the value of the corresponding attribute from SETUP in PLAN. If no method is sent in the PLAN message, the symbol NONE will be returned.

10. Attributes for BUILD_SETUP in NOTPLANNED

The NOTPLANNED BUILD_SETUP message is returned when there is an error planning a setup. The attributes of this message describe the error that occurred.

NAME — The value of this attribute is a symbol which is the name of the setup that was not successfully planned. Its value is the value of the attribute NAME for SETUP in PLAN.

ERRORS — The value of this attribute is a string describing the error that occurred in attempting to create a plan. See the section on Errors and Warnings for a list of strings.

11. Attributes for OBJECT in COPY

The COPY OBJECT message updates the current setup in the modeler with the fixtures that will hold the part. The attributes of this message name the fixtures from a standard fixturing library and the current setup environment. Each type OBJECT featurelist in a COPY sentence contains:

NAME — The value of this attribute is a symbol which is the name of a fixture that is to be copied from the fixture library to the current part environment. These names have the form Name_Number where Name is one of LOCATOR, TOE_CLAMP, FIXED_VISE_JAW, MOVEABLE_VISE_JAW, PARALLEL_BAR, RISER, BOLT, SUBPLATE. Each type of fixture is uniquely identified by appending a number to its name. If the fixture library has three locators, they would be referenced by LOCATOR_1, LOCATOR_2, and LOCATOR_3.

APPLICATION — The value of this attribute is a symbol which is the name of the application in the modeler which has the fixture library. In the current version of HX this is always LIBRARY.

ENVIRONMENT — The value of this attribute is a symbol which is the name of the environment in the modeler which has the fixture library. In the current version of HX this is always FIXELS.

TO_APPLICATION — The value of this attribute is a symbol which is the name of current part application in the modeler. The value used is the value of the APPLICATION attribute from the PLAN verb.

TO_ENVIRONMENT — The value of this attribute is a symbol which is the name of current part environment in the modeler. The value used is the value of the ENVIRONMENT attribute from the PLAN verb.

12. Attributes for OBJECT in TRANSFORM

The TRANSFORM OBJECT message updates the position of the fixtures (created by the COPY OBJECT message) in the current setup in the modeler. The COPY OBJECT message cannot position the fixtures, so the TRANSFORM message is used to correct the positions. The attributes of this message name the fixtures and their locations. Each type OBJECT featurelist in

a TRANSFORM sentence contains:

NAME	The value of this attribute is a symbol which is the name of a fixture that is to be translated to its final position. See the NAME attribute for OBJECT in COPY for a description of its values.
APPLICATION	The value of this attribute is a symbol which is the name of current part application in the modeler. The value used is the value of the APPLICATION attribute from the PLAN verb. Obviously this is the same as the value of the TO_APPLICATION in the corresponding COPY sentence.
ENVIRONMENT	The value of this attribute is a symbol which is the name of current part environment in the modeler. The value used is the value of the ENVIRONMENT attribute from the PLAN verb. Obviously this is the same as the value of the TO_ENVIRONMENT in the corresponding COPY sentence.
Z_ROTATION	The value of this attribute is an real in units of degrees that is the Z axis rotation to be applied to the fixture object to correctly orient it in the current part model.
TRANSLATION	The value of this attribute is a vector of 3 reals that is the translations to be applied to the fixture object to correctly position it in the current part model.

13. Sample Sentences

Listed below are four sample files tool.dc.1, tool.dc.2, tool.dc.3, and tool.dc.4 approximating the four setups for an IMW test part.

File tool.dc.1

On the first setup just simple face milling on the top surface is done.

```
(plan ((type setup)(name fool)
       (part obj1)(application app1)(environment env1)
       (finished_part none)
       (method none)
       (setup_no 1)
       (angle 0)
       (subplate_depth 0)
       (translation (0 0 1.25))
       (x_rotation -90)
       (y_rotation 0)
       (z_rotation 0)
       (major_normal (0 1 0))
       (major_pos 0)
       (minor_normal (0 1 0))   ; variable
       (minor_pos 0)
      )
      ((type tool_path)(name tool1)
       (rapid (-.5 .25 1.135))
      )
      ((type tool_path)(name tool1)    ; facemill
       (sfm 300)
       (horsepower 3)
       (diameter 0.6)
```

```
        (linear (3 .25 1.135))))
```

File tool.dc.2

On the second setup the part is flipped over and the other face is milled and exposed edges are edge milled. A channel or slot in the center of the part is milled.

```
(plan ((type setup)(name fool)
       (part obj2)(application app2)(environment env2)
       (finished_part none)
       (method none)
       (setup_no 2)
       (angle 0)
       (subplate_depth 0)
       (translation (0 0 0))
       (x_rotation 0)
       (y_rotation 0)
       (z_rotation 0)
       (major_normal (0 1 0))
       (major_pos 0)
       (minor_normal (0 -1 0))     ; variable
       (minor_pos 0)
       )
      ((type tool_path)(name tool1)
       (rapid (-.2 -.5 -.5))
      )
      ((type tool_path)(name tool1)   ; endmill
       (sfm 300)
       (horsepower 3)
       (diameter 0.6)
       (linear (-.2 1.5 -.5)))
      ((type tool_path)(name tool2)
       (rapid (-.5 .6 .42))
      )
      ((type tool_path)(name tool2)   ; facemill
       (sfm 300)
       (horsepower 3)
       (diameter 1.5)
       (linear (3 .6 .42)))
      ((type tool_path)(name tool3)
       (rapid (1.25 -.5 .25))
      )
      ((type tool_path)(name tool3)   ; millslot
       (sfm 300)
       (horsepower 3)
       (diameter 1.5)
       (linear (1.25 1.5 .25)))
)
```

File tool.dc.3

On the third setup the remaining end and face milling is done and two holes are drilled.

```
(plan ((type setup)(name fool)
       (part obj3)(application app3)(environment env3)
       (finished_part none)
       (method none)
       (setup_no 3)
       (angle 0)
```

```
        (subplate_depth 0)
        (translation (0 0 0))
        (x_rotation 0)
        (y_rotation 0)
        (z_rotation 0)
        (major_normal (0 1 0))
        (major_pos 0)
        (minor_normal (0 -1 0))     ; variable
        (minor_pos 0)
       )
      ((type tool_path)(name tool1)
       (rapid (-.2 -.5 -.5))
      )
      ((type tool_path)(name tool1)   ; endmill
       (sfm 300)
       (horsepower 3)
       (diameter 0.6)
       (linear (-.2 1.5 -.5)))
      ((type tool_path)(name tool2)
       (rapid (0.25 .57 .34))
      )
      ((type tool_path)(name tool2)    ; facemill
       (sfm 300)
       (horsepower 3)
       (diameter .3)
       (linear (3 .6 .34)))
      ((type tool_path)(name tool3)
       (rapid (.25 .6 .34))
      )
      ((type tool_path)(name tool3)    ; drill 1
       (sfm 300)
       (horsepower 3)
       (diameter 1.5)
       (linear (.25 .6 -.5)))
      ((type tool_path)(name tool4)
       (rapid (2.25 .6 .34))
      )
      ((type tool_path)(name tool4)    ; drill 2
       (sfm 300)
       (horsepower 3)
       (diameter .3)
       (linear (2.25 .6 -.5)))
)
```

File tool.dc.4

On the fourth (and final) setup the part is mounted on a sine table at 30 degrees. The edge overhanging the table is both end milled and face milled to produce a beveled edge.

```
(plan ((type setup)(name foo4)
       (part obj4)(application app4)(environment env4)
       (finished_part none)
       (method none)
       (setup_no 4)
       (angle 30)
       (position x)
       (subplate_depth 0)
       (translation (0 0 .5))
```

```
        (x_rotation -30)
        (y_rotation 0)
        (z_rotation 0)
        (major_normal (0 1 0))
        (major_pos 0)
        (minor_normal (0 0.8660 -.5))  ; variable
        (minor_pos 0)
       )
      ((type tool_path)(name tool1)
       (rapid ( -.5 .2 .8))
      )
      ((type tool_path)(name tool1)   ; facemill
       (sfm 300)
       (horsepower 3)
       (diameter 0.6)
       (linear (2.8 .2 .8)))
      ((type tool_path)(name tool2)
       (rapid (-.5 -0.1 .2))
      )
      ((type tool_path)(name tool2)   ; endmill
       (sfm 300)
       (horsepower 3)
       (diameter 0.6)
       (linear (2.8 -0.1 .2))
      )
)
```

After loading in the fixture library and loading in the part into APP/ENV/OBJ 1 through 4 in the modeling expert MX, the above files can be read into the holding expert HX to produce the following output: (The user input is underlined. Some long lists of part coordinates have been omitted from the output.)

```
% hx

> Starting HX version 0.6


> (read ((type file)(name "tool.dc.1")))

(Holding) Error in Setup::AttrMethod: bad method NONE received
(Output::send) '(GET ((NAME G0) (TYPE MESSAGE) (TO mx) (FROM hx) ) ((T
YPE BREP) (APPLICATION APP1) (ENVIRONMENT ENV1) (OBJECT OBJ1) (NAME OB
J1) ) )' (131)

 Fixture plan with toeclamps or a vise may exist
xcmin xcmax xcmin xcmax   0.000   2.750   0.000   1.085
xvmin xvmax -10.000  10.000

 *** Fixture with a vise ***
xdist from left end of solid jaw = -0.375
distance between jaws =  0.700

 *** No parallel bar is needed ***
       Name              Size        X          Y          Z
-------------------------------------------------------------
     fixed_vise_jaw                 5.868      3.293      0.000
               part                 4.493      4.493      0.000
```

```
   moveable_vise_jaw                5.868       5.193       0.000


(Output::send) '(COPY ((NAME G1) (TYPE MESSAGE) (TO mx) (FROM hx) ) ((
TYPE OBJECT) (NAME FIXED_VISE_JAW_1) (APPLICATION LIBRARY) (ENVIRONMEN
T FIXELS) (TO_APPLICATION APP1) (TO_ENVIRONMENT ENV1) ) ((TYPE OBJECT)
 (NAME MOVEABLE_VISE_JAW_1) (APPLICATION LIBRARY) (ENVIRONMENT FIXELS)
 (TO_APPLICATION APP1) (TO_ENVIRONMENT ENV1) ) )' (312)
(Output::send) '(TRANSFORM ((NAME G2) (TYPE MESSAGE) (TO mx) (FROM hx)
 ) ((TYPE OBJECT) (NAME FIXED_VISE_JAW_1) (APPLICATION APP1) (ENVIRONM
ENT ENV1) (Z_ROTATION 0.000000) (TRANSLATION (5.867600 3.292600 0.0000
00 )) ) ((TYPE OBJECT) (NAME MOVEABLE_VISE_JAW_1) (APPLICATION APP1) (
ENVIRONMENT ENV1) (Z_ROTATION 0.000000) (TRANSLATION (5.867600 5.19260
0 0.000000 )) ) )' (351)
(Output::send) '(PLANNED ((NAME NONE) (TYPE MESSAGE) (TO none) (FROM n
one) ) ((NAME FOO1) (TYPE BUILD_SETUP) (X_ROTATION 0.000000) (Y_ROTATI
ON 0.000000) (Z_ROTATION 180.000000) (TRANSLATION (0.000000 0.000000 0
.000000 )) (SETUP_NO 1) (APPLICATION APP1) (ENVIRONMENT ENV1) (PART OB
J1) (FINISHED_PART NONE) (ANGLE 0.000000) (SUBPLATE_DEPTH 0.000000) (P
OSITION X) (MODE MANUAL) (METHOD NONE) (NC_FILENAME "/ssdh/usr2/baird/
imw/hx/doc/example/hx_macro_1") (FIXTURES (FIXED_VISE_JAW_1 MOVEABLE_V
ISE_JAW_1 )) ) )' (490)
no such name 'none' for Send


> (read ((type file)(name "tool.dc.2")))

(Holding) Error in Setup::AttrMethod: bad method NONE received
(Output::send) '(GET ((NAME G3) (TYPE MESSAGE) (TO mx) (FROM hx) ) ((T
YPE BREP) (APPLICATION APP2) (ENVIRONMENT ENV2) (OBJECT OBJ2) (NAME OB
J2) ) )' (131)

 Fixture plan with toeclamps or a vise may exist
xcmin xcmax zcmin zcmax   0.300   2.700   0.000   0.200
xvmin xvmax   0.300  10.000

 *** Fixture with a vise ***
xdist from left end of solid jaw = -0.300
distance between jaws =  1.335

 *** A parallel bar is needed ***
    maximum length=  2.300
    maximum width=  1.035
    minimum height=  0.738

       Name                Size        X           Y           Z
-------------------------------------------------------------------
      fixed_vise_jaw                  5.793       3.293       0.000
        parallel_bar                  4.493       4.493       0.000
                part                  4.493       4.493       0.000
  moveable_vise_jaw                   5.793       5.828       0.000


(Holding) Warning in SendPlan: no tranformation matrix
(Output::send) '(COPY ((NAME G4) (TYPE MESSAGE) (TO mx) (FROM hx) ) ((
```

```
TYPE OBJECT) (NAME FIXED_VISE_JAW_1) (APPLICATION LIBRARY) (ENVIRONMEN
T FIXELS) (TO_APPLICATION APP2) (TO_ENVIRONMENT ENV2) ) ((TYPE OBJECT)
 (NAME PARALLEL_BAR_1) (APPLICATION LIBRARY) (ENVIRONMENT FIXELS) (TO_
APPLICATION APP2) (TO_ENVIRONMENT ENV2) ) ((TYPE OBJECT) (NAME MOVEABL
E_VISE_JAW_1) (APPLICATION LIBRARY) (ENVIRONMENT FIXELS) (TO_APPLICATI
ON APP2) (TO_ENVIRONMENT ENV2) ) )' (438)
(Output::send) '(TRANSFORM ((NAME G5) (TYPE MESSAGE) (TO mx) (FROM hx)
 ) ((TYPE OBJECT) (NAME FIXED_VISE_JAW_1) (APPLICATION APP2) (ENVIRONM
ENT ENV2) (Z_ROTATION 0.000000) (TRANSLATION (5.792600 3.292600 0.0000
00 )) ) ((TYPE OBJECT) (NAME PARALLEL_BAR_1) (APPLICATION APP2) (ENVIR
ONMENT ENV2) (Z_ROTATION 0.000000) (TRANSLATION (4.492600 4.492600 0.0
00000 )) ) ((TYPE OBJECT) (NAME MOVEABLE_VISE_JAW_1) (APPLICATION APP2
) (ENVIRONMENT ENV2) (Z_ROTATION 0.000000) (TRANSLATION (5.792600 5.82
7600 0.000000 )) ) )' (494)
(Output::send) '(PLANNED ((NAME NONE) (TYPE MESSAGE) (TO none) (FROM n
one) ) ((NAME FOO1) (TYPE BUILD_SETUP) (X_ROTATION 0.000000) (Y_ROTATI
ON 0.000000) (Z_ROTATION 0.000000) (TRANSLATION (0.000000 0.000000 0.0
00000 )) (SETUP_NO 2) (APPLICATION APP2) (ENVIRONMENT ENV2) (PART OBJ2
) (FINISHED_PART NONE) (ANGLE 0.000000) (SUBPLATE_DEPTH 0.000000) (POS
ITION X) (MODE MANUAL) (METHOD NONE) (NC_FILENAME "/esdh/usr2/baird/im
w/hx/doc/example/hx_macro_2") (FIXTURES (FIXED_VISE_JAW_1 PARALLEL_BAR
_1 MOVEABLE_VISE_JAW_1 )) ) )' (503)
no such name 'none' for Send


> (read ((type file)(name "tool.dc.3")))

(Holding) Error in Setup::AttrMethod: bad method NONE received
(Output::send) '(GET ((NAME G6) (TYPE MESSAGE) (TO mx) (FROM hx) ) ((T
YPE BREP) (APPLICATION APP3) (ENVIRONMENT ENV3) (OBJECT OBJ3) (NAME OB
J3) ) )' (131)

 Fixture plan with toeclamps or a vise may exist
xcmin xcmax zcmin zcmax   0.300   2.600   0.000   0.290
xvmin xvmax   0.300  10.000

 *** Fixture with a vise ***
xdist from left end of solid jaw = -0.300
distance between jaws =  1.335

 *** A parallel bar is needed ***
    maximum length=  2.200
    maximum width=  1.035
    minimum height=  0.647

        Name                Size        X          Y          Z
-------------------------------------------------------------------
      fixed_vise_jaw                  5.793      3.293      0.000
        parallel_bar                  4.493      4.493      0.000
                part                  4.493      4.493      0.000
   moveable_vise_jaw                  5.793      5.828      0.000


(Holding) Warning in SendPlan: no tranformation matrix
(Output::send) '(COPY ((NAME G7) (TYPE MESSAGE) (TO mx) (FROM hx) ) ((
TYPE OBJECT) (NAME FIXED_VISE_JAW_1) (APPLICATION LIBRARY) (ENVIRONMEN
```

```
T FIXELS) (TO_APPLICATION APP3) (TO_ENVIRONMENT ENV3) ) ((TYPE OBJECT)
 (NAME PARALLEL_BAR_1) (APPLICATION LIBRARY) (ENVIRONMENT FIXELS) (TO_
APPLICATION APP3) (TO_ENVIRONMENT ENV3) ) ((TYPE OBJECT) (NAME MOVEABL
E_VISE_JAW_1) (APPLICATION LIBRARY) (ENVIRONMENT FIXELS) (TO_APPLICATI
ON APP3) (TO_ENVIRONMENT ENV3) ) )' (438)
(Output::send) '(TRANSFORM ((NAME G8) (TYPE MESSAGE) (TO mx) (FROM hx)
 ) ((TYPE OBJECT) (NAME FIXED_VISE_JAW_1) (APPLICATION APP3) (ENVIRONM
ENT ENV3) (Z_ROTATION 0.000000) (TRANSLATION (5.792600 3.292600 0.0000
00 )) ) ((TYPE OBJECT) (NAME PARALLEL_BAR_1) (APPLICATION APP3) (ENVIR
ONMENT ENV3) (Z_ROTATION 0.000000) (TRANSLATION (4.492600 4.492600 0.0
00000 )) ) ((TYPE OBJECT) (NAME MOVEABLE_VISE_JAW_1) (APPLICATION APP3
) (ENVIRONMENT ENV3) (Z_ROTATION 0.000000) (TRANSLATION (5.792600 5.82
7600 0.000000 )) ) )' (494)
(Output::send) '(PLANNED ((NAME NONE) (TYPE MESSAGE) (TO none) (FROM n
one) ) ((NAME FOO1) (TYPE BUILD_SETUP) (X_ROTATION 0.000000) (Y_ROTATI
ON 0.000000) (Z_ROTATION 0.000000) (TRANSLATION (0.000000 0.000000 0.0
00000 )) (SETUP_NO 3) (APPLICATION APP3) (ENVIRONMENT ENV3) (PART OBJ3
) (FINISHED_PART NONE) (ANGLE 0.000000) (SUBPLATE_DEPTH 0.000000) (POS
ITION X) (MODE MANUAL) (METHOD NONE) (NC_FILENAME "/ssdh/usr2/baird/im
w/hx/doc/example/hx_macro_3") (FIXTURES (FIXED_VISE_JAW_1 PARALLEL_BAR
_1 MOVEABLE_VISE_JAW_1 )) ) )' (503)
no such name 'none' for Send
```

> (read ((type file)(name "tool.dc.4")))

```
(Holding) Error in Setup::AttrMethod: bad method NONE received
(Output::send) '(GET ((NAME G9) (TYPE MESSAGE) (TO mx) (FROM hx) ) ((T
YPE BREP) (APPLICATION APP4) (ENVIRONMENT ENV4) (OBJECT OBJ4) (NAME OB
J4) ) )' (131)

 Fixture plan with toeclamps or a vise may exist


*** Pin Location in part coord. ***

PIN       X      Y     Max. Height
 1     0.457 -0.250   0.240
 2     1.871 -0.250   0.240
 3    -0.250  0.457   0.240


Total no. of Clamp Ranges = 3

                *** CLAMP RANGES ***
     x1      y1      z1      x2      y2      z2   length
1  0.500   0.000   0.170   2.000   0.000   0.170   1.500
2  0.000   0.000   0.340   0.500   0.000   0.340   0.500
3  2.000   0.000   0.340   2.500   0.000   0.340   0.500

 *** Fixture with toeclamps on a sine_plate (angle =  30.000) ***
        Name                Size          X           Y           Z
-------------------------------------------------------------------
                     pin                 13.935      27.370      0.000
                     pin                 15.349      27.370      0.000
                     pin                 13.228      28.077      0.000
                    part                 13.478      27.620      0.000
```

```
clamp                14.278      27.620      0.170
clamp                15.178      27.620      0.170


(Output::send) '(COPY ((NAME G10) (TYPE MESSAGE) (TO mx) (FROM hx) ) (
(TYPE OBJECT) (NAME LOCATOR_1) (APPLICATION LIBRARY) (ENVIRONMENT FIXE
LS) (TO_APPLICATION APP4) (TO_ENVIRONMENT ENV4) ) ((TYPE OBJECT) (NAME
 LOCATOR_2) (APPLICATION LIBRARY) (ENVIRONMENT FIXELS) (TO_APPLICATION
 APP4) (TO_ENVIRONMENT ENV4) ) ((TYPE OBJECT) (NAME LOCATOR_3) (APPLIC
ATION LIBRARY) (ENVIRONMENT FIXELS) (TO_APPLICATION APP4) (TO_ENVIRONM
ENT ENV4) ) ((TYPE OBJECT) (NAME TOE_CLAMP_1) (APPLICATION LIBRARY) (E
NVIRONMENT FIXELS) (TO_APPLICATION APP4) (TO_ENVIRONMENT ENV4) ) ((TYP
E OBJECT) (NAME TOE_CLAMP_2) (APPLICATION LIBRARY) (ENVIRONMENT FIXELS
) (TO_APPLICATION APP4) (TO_ENVIRONMENT ENV4) ) )' (663)
(Output::send) '(TRANSFORM ((NAME G11) (TYPE MESSAGE) (TO mx) (FROM hx
) ) ((TYPE OBJECT) (NAME LOCATOR_1) (APPLICATION APP4) (ENVIRONMENT EN
V4) (Z_ROTATION 0.000000) (TRANSLATION (13.934900 27.369801 0.000000 )
) ) ((TYPE OBJECT) (NAME LOCATOR_2) (APPLICATION APP4) (ENVIRONMENT EN
V4) (Z_ROTATION 0.000000) (TRANSLATION (15.349100 27.369801 0.000000 )
) ) ((TYPE OBJECT) (NAME LOCATOR_3) (APPLICATION APP4) (ENVIRONMENT EN
V4) (Z_ROTATION 0.000000) (TRANSLATION (13.227800 28.076900 0.000000 )
) ) ((TYPE OBJECT) (NAME TOE_CLAMP_1) (APPLICATION APP4) (ENVIRONMENT
ENV4) (Z_ROTATION 0.000000) (TRANSLATION (14.277800 27.619801 0.170000
 )) ) ((TYPE OBJECT) (NAME TOE_CLAMP_2) (APPLICATION APP4) (ENVIRONMEN
T ENV4) (Z_ROTATION 0.000000) (TRANSLATION (15.177800 27.619801 0.1700
00 )) ) )' (763)
(Output::send) '(PLANNED ((NAME NONE) (TYPE MESSAGE) (TO none) (FROM n
one) ) ((NAME FOO4) (TYPE BUILD_SETUP) (X_ROTATION 0.000000) (Y_ROTATI
ON 0.000000) (Z_ROTATION 180.000000) (TRANSLATION (0.000000 0.000000 0
.000000 )) (SETUP_NO 4) (APPLICATION APP4) (ENVIRONMENT ENV4) (PART OB
J4) (FINISHED_PART NONE) (ANGLE 30.000000) (SUBPLATE_DEPTH 0.000000) (
POSITION X) (MODE MANUAL) (METHOD NONE) (NC_FILENAME "/ssdh/usr2/baird
/imw/hx/doc/example/hx_macro_4") (FIXTURES (LOCATOR_1 LOCATOR_2 LOCATO
R_3 TOE_CLAMP_1 TOE_CLAMP_2 )) ) )' (508)
no such name 'none' for Send
```

14. Errors and Warnings

If the holding expert cannot plan fixtures for the setup, it will return a NOTPLANNED message to the originator of the PLAN request. See the NOTPLANNED verb description for a list attributes. The ERRORS attribute will have one of the following strings (underlined):

"No fixture element"
This means that the main fixture planning routine could not create a plan. One frequent cause of this is that the part is too small or too big for the current fixture geometry compiled into holding expert.

"No macro file name"
This means that although a plan was created, the nc file name containing the code to load the fixtures was not generated. This case only happens when there is an internal error in the main fixture planning routine.

"No part brep from modeler"

This means that the GET message to the modeling expert returned NOTGOTTEN. This typically happens when the PART, APPLICATION, or ENVIRONMENT passed in from the PLAN verb are not the names that are in the modeler.

"Unable to copy fixtures to model"
This means that after creating a setup plan containing a list of fixtures, the COPY message (from the holding expert to the modeler) failed to copy fixtures from the fixture library to the current part environment. Typically this is caused by not loading the fixture library into the modeler, or the fixture library names do not agree with the holding expert names, or that the number of fixtures in the library is less than what the holding expert requested.

"Unable to transform fixtures in model"
This means that after creating a setup plan containing a list of fixtures and copying them from the library to the current part environment, the TRANSFORM message (from the holding expert to the modeler) failed to transform fixtures in the current part environment. This has the same failures as the previous "Unable to copy fixtures to model" message but is less likely to appear because the copy message generates the error first. This message will be seen if the modeler copy command silently fails, but the transform reports NOTTRANSFORMED.

Other errors and warning are just printed on the terminal. Since the other expert systems will not utilize warnings, they are printed instead of being included in a return message. The most common causes of warnings are using an obsolete attribute or omitting an attribute (but a reasonable default value can be supplied.) Other errors printed out instead of returning NOTPLANNED are fatal. These are typically from leaving out an attribute that no reasonable default can be supplied for (e.g., MINOR_NORMAL.) Any message with "new TYPENAME failed" is a fatal error in memory allocation procedure (e.g., malloc out of memory.) The only option at this point is to "(quit)" and restart the program.

Both printed errors and warnings have the following format:

```
(Holding) ETYPE in PROC: ERRSTR
```

where ETYPE is "Error" or "Warning", PROC is a procedure name or a class::method name and ERRSTR is some descriptive string. Note! All errors are printed on stdout, not stderr!

Center for
Integrated Manufacturing Decision Systems

Generic Environment for Unix-based Experts

Duane T. Williams

March, 1990

Abstract:

This document describes a library of C++ classes that supports the common elements of the Cutting, Holding, Sensing, and Modeling subsystems of the IMW (Intelligent Machining Workstation). Enough information is provided to enable a programmer to develop or maintain one of these programs.

A brief description of the general form of the Unix-based subsystems is given, including an introduction to the C++ Task System. The bulk of the document describes the class interfaces to various components of the internal representation of FEL sentences.

Copyright © 1990 Carnegie Mellon University

Contact:

David Bourne
CIMDS
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-8810

1. Introduction

This document describes a library of C++ classes that supports the common elements of the Cutting, Holding, Sensing, and Modeling subsystems of the Intelligent Machining Workstation (IMW), and how to construct such a subsystem based on this library.[1] Since the various subsystems that have been developed with this library are termed "experts", e.g., Cutting Expert, we call the library itself the Generic Expert.

This document is necessarily somewhat technical, since it is intended for someone writing a computer program such as the Cutting Expert in the C++ language. It presupposes an understanding of C++.[2]

2. Purpose

The purpose of the Generic Expert library is to reduce the effort required to produce a subsystem for the IMW that interfaces properly with the rest of the system. Since communication capabilities are required by all subsystems, they are provided in the Generic Expert. All subsystems are also expected to communicate in a common language, Feature Exchange Language (FEL), so an FEL parser and FEL sentence generation procedures are part of the Generic Expert. A simple terminal interface is also provided.

In the following sections, we describe the various user-accessible C++ classes in the Generic Expert library that enable the user to take advantage of these common capabilities. We also recommend a programming style that will enable the user to best take advantage of possible future enhancements.

3. C++ Task System

The implementation of the Generic Expert is based on the C++ Task System, which is supplied as a standard library with the AT&T C++ translator. We describe only the basic features in this section.[3] Note: we use the version of the Task System which supports waiting on UNIX SIGNALS, as described in [Shopiro 1987]. The Generic Expert will not work with the old version.

3.1. Tasks

The Task System defines a type of "lightweight process" in the guise of the **class task**. Constructors of classes (directly) derived from class task can share the CPU under the control of a task scheduler. Each such task can be suspended and later resumed without disturbing its internal state, but suspension is always voluntary, occurring only when a task chooses to wait for an event to occur.

Tasks are created by defining a class that derives directly from the base class task and by implementing a constructor for the new class that does the work that the new task is supposed to do. The only serious requirement that the constructor must satisfy is that it must terminate properly by invoking the function **resultis(0)**. The example near the

1. The library file is named libshell.a. The system also requires the new AT&T C++ Task System that supports software interrupts (i.e., signals).
2. See [Stroustrup 1986].
3. See [Stroustrup 1987] and [Shopiro 1987] for detailed descriptions of the Task System.

end of this document does this by using the macro TERMINATE (defined in macros.h).

3.2. Queues

The Task System defines a type of first-in-first-out queue based on two classes: qtail and **qhead**. Tasks can communicate through such a queue. One task "puts" an object to an instance of class qtail, while another task waits for the arrival of an object on the corresponding qhead. The action of waiting causes the task to be swapped out by the task scheduler; the arrival of a object on the qhead causes tasks that are waiting on that qhead to be put on the run chain.

3.2.1. Message Queues

Queues are used extensively within the Generic Expert for intra-expert communication between the multiple tasks that make up a program built from the Generic Expert library. These queues are instances of two classes, **MessageQHead** and **MessageQTail**, that are derived from qhead and qtail. These classes are specialized to handle only objects of type Message.

Class MessageQTail defines only two functions:

```
int put (Message*)
MessageQHead* head ()
```

The *put* function puts a message on a queue; this action can cause a task to block if the queue is full. The *head* function returns a pointer to the head of the queue.

Class MessageQHead defines three functions:

```
Message* get ()
int putback (Message*)
MessageQTail* tail ()
```

The *get* function retrieves the next message from the head of a queue; if there is no message, the task will block until a message arrives. The *putback* function returns a message to the head of a queue. The *tail* function returns a pointer to the tail of the queue.

3.2.2. Creating Queues between Tasks

The following example shows how to connect two tasks via a queue. This code is taken from the "central control" task in the Generic Expert; this is how the Generic Expert sets up queues to communicate with the user's Expert task.

```
/* create the queue I will read from */
MessageQHead* myQHead = new MessageQHead;
MessageQTail* myQTail = myQHead -> tail ();

/* create the queue he will read from */
MessageQHead* aQHead = new MessageQHead;
MessageQTail* toExpert = aQHead -> tail ();

/* create the task */
```

```
(void) new Expert (argv0, aQHead, myQTail);
```

Note that only one end of a queue (here a MessageQHead) is actually created using *new*. You get a pointer to the other end by invoking the appropriate *tail* or *head* function (in our case, *tail*). The task Expert, defined in expert.h, has a MessageQHead* parameter, from which it receives its messages, and a MessageQTail* parameter, to which it puts messages that it wants to send to the "central control" task. The "central control" task sends messages to Expert by putting them on the *toExpert* MessageQTail.

The user's Expert task could create sub-tasks of its own, together with appropriate queues to communicate with them.

4. Class Expert

The primary effort in developing a program based on the Generic Expert is the implementation of the constructor for class Expert, as declared in the file **expert.h**. What this means, basically, is that one has to write a C++ subroutine with the following procedure declaration:

```
Expert::Expert (char* myName,
                MessageQHead* readFrom,
                MessageQTail* writeTo)
```

This subroutine automatically becomes a task[4] within the final program. It communicates with other tasks and subsystems via two message queues, which are capable of transmitting Sentences, described below. Incoming messages are retrieved from the head of the "readFrom" queue; outgoing messages are placed on the tail of the "writeTo" queue.

To wait for the arrival of a message, use the member function *get*:

```
/* Messages are described in the next section */
Message* request;
request = readFrom -> get();
```

This puts your task to sleep until a message is available from the "readFrom" queue. To send a message, use the member function *put*:

```
/* Assume that aSentence is an instance of class Sentence,
   described below. */
Message* reply = new Message (OUTPUT, aSentence);
writeTo -> put (reply);
```

This creates a new message with a Sentence that is destined for a subsystem on the network and then puts it into a queue where it can be picked up by another task, which will handle the details of the transmission. Sending a message does not suspend a task.

A prototype implementation of Expert::Expert is provided in the file **expert.c**. The developer should replace the program block under case INPUT with his own code. See the Example section later in the document. One of the main features of this prototype is that it gets all input in one place. For this to work effectively, the code you add should

4. See [Stroustrup 1987].

be designed to do only a small amount of work each time this task becomes active.[5]

5. Class Message

Instances of this class represent messages that are passed between tasks within a single subsystem; so users do not have to know much about them, except for how to retrieve the contents—presumably a Sentence—from an incoming message and how to package a Sentence into a message for transmission to another subsystem.

Class Message contains a public data field called mContents which contains a pointer to the contents of the message. No information about the size of the data is maintained by the Message class. Here is how a sentence can be extracted from a message.

```
/* get a message and access its contents as a Sentence */
Message* request;
request = readFrom -> get();
Sentence* theSentence = (Sentence*)(request -> mContents);
```

Packaging a sentence as a message for transmission to the Output task for delivery to another subsystem is simple using the constructor for class Message:

```
Message* reply = new Message (OUTPUT, aSentence);
writeTo -> put (reply);
```

Messages can also be used to put an error message on the standard output stream as follows:

```
Message* error = new Message (ERROR, "Naughty, naughty!");
writeTo -> put (error);
```

6. Class Sentence

Instances of class Sentence represent FEL messages. Member functions provide access to significant components of a message and allow the construction of new messages. Currently supported functions include:

Sentence ()
Sentence (VERB_ENUM aVerb)
The constructor takes an optional VERB_ENUM as an argument.[6] If none is supplied, the verb will default to VERB_NONE.

Sentence* append (FeatureList* fList)
The function *append* appends a FeatureList to a Sentence. The argument is a pointer to the FeatureList to be appended. A pointer to the new Sentence is returned.

ADDR_ENUM destination ()

5. We do not describe in this document how to accomplish this. Future documentation on the to-be-developed inter-subsystem dialogue mechanism will include a technique based on the rescheduling of a task.
6. Refer to the file verb.h for the definition of VERB_ENUM.

The function *destination* returns as an ADDR_ENUM the address of the receiver of the message.[7] This is the value of the TO attribute within the feature list of type MESSAGE.

FeatureList* featureListOf (TYPE_ENUM aType)
The function *featureListOf* searches the feature lists that compose a Sentence and returns a pointer to the first feature list whose TYPE attribute has the value aType.

Sentence* insert (FeatureList* fList)
The function *insert* inserts a FeatureList before other FeatureLists in a Sentence. The argument is a pointer to the FeatureList to be inserted. A pointer to the new Sentence is returned.

Sentence* reply (VERB_ENUM aVerb)
The function *reply* creates and returns a pointer to a new Sentence that can be used as a reply to a message. It will contain a FeatureList with the same NAME and TYPE attributes as the original Sentence, but with values of the TO and FROM attributes reversed. The argument aVerb specifies the value of the verb for the new message.

Sentence* setVerb (VERB_ENUM aVerb)
The function *setVerb* makes the argument aVerb the new value of the verb of the Sentence. A pointer to the Sentence is returned.

ADDR_ENUM source ()
The function *source* returns as an ADDR_ENUM the address of the sender of the message. This is the value of the FROM attribute within the feature list of type MESSAGE.

VERB_ENUM verb ()
The function *verb* returns the verb of the Sentence as a VERB_ENUM.

The intended use of this class is illustrated by the following example of how to create a reply to a message.

```
{
   /* assume that aSentence is a pointer to a Sentence          */
   /* and that aVerb is appropriate for the reply to aSentence */
   Sentence* response = aSentence -> reply (aVerb);

   /* add one FeatureList */
   FeatureList* fList = new FeatureList;
   fList -> include (ATTR_NAME, VT_SYMBOL, aName);
   fList -> include (ATTR_TYPE, VT_TYPE, TYPE_FIXTURE);
   ...
   response -> append (fList);

   /* send the response */
   /* assume that writeTo is an appropriate MessageQTail
      pointer */
```

7. Refer to the file addr.h for the definition of ADDR_ENUM.

```
    Message* m = new Message (OUTPUT, response);
    writeTo -> put (m);
}
```

7. Class Sentence Iterator

Instances of class SentenceIterator allow one to step through the feature lists of a Sentence. Each call to the member function next() returns a pointer to a FeatureList. A zero is returned after all the feature lists have been returned. Currently supported functions include:

SentenceIterator (Sentence& aSentence)
The constructor takes a reference to a Sentence as its sole argument.

FeatureList* first ()
The function *first* resets the iterator and returns a pointer to the first FeatureList in the Sentence.

FeatureList* next ()
The function *next* returns a pointer to the FeatureList following the last one returned by this function, or to the first FeatureList in the Sentence if none has been returned from the SentenceIterator since it was created. A zero is returned when a pointer to every FeatureList has been returned.

This class is provided so that one can sequentially process all the feature lists in a message without having to know the internal representation of messages. The intended use is illustrated in the following example:

```
{
   /* assume that aSentence is a pointer to a Sentence */

   SentenceIterator sIter (*aSentence);
   FeatureList* fList;

   while (fList = sIter.next()) {
      /* process the FeatureList pointed to by fList */
      ...
   }
}
```

8. Class Feature List

Instances of class FeatureList represent the list of attribute/value pairs that are the principal components of FEL messages. Member functions provide access to the values of the pairs and allow the construction of new lists of pairs. Currently supported functions include:

FeatureList ()
FeatureList (char* aName)

The constructor takes an optional char* as an argument. If none is supplied, the value will default to zero. When the optional name is supplied, it is construed as the name of an existing FeatureList and no attribute/value pairs should be added to this new FeatureList.

FeatureList* append (AttributeValuePair* avPair)
The function *append* appends an AttributeValuePair to a FeatureList. The argument is a pointer to the AttributeValuePair to be appended. A pointer to the FeatureList is returned.

void include (ATTR_ENUM, VT_ENUM, int, int)
void include (ATTR_ENUM, VT_ENUM, long, int)
void include (ATTR_ENUM, VT_ENUM, float, int)
void include (ATTR_ENUM, VT_ENUM, double, int)
void include (ATTR_ENUM, VT_ENUM, char*)
void include (ATTR_ENUM, VT_ENUM, List*)
void include (ATTR_ENUM, VT_ENUM, char**)
void include (ATTR_ENUM, VT_ENUM, VECTOR)
The *include* functions add attribute/value pairs to the beginning of a FeatureList. The attribute is given in the first argument. The type of value is given in argument two and the actual value in argument three. An optional fourth argument specifies the units (as a UNIT_ENUM) of the value.

FeatureList* insert (AttributeValuePair* avPair)
The function *insert* inserts an AttributeValuePair before other AttributeValuePairs in a FeatureList. The argument is a pointer to the AttributeValuePair to be inserted. A pointer to the FeatureList is returned.

TYPE_ENUM typeOf ()
The function *typeOf* returns the value of the pair whose attribute is ATTR_TYPE.

ValueType* valueOfAttribute (ATTR_ENUM)
ValueType* valueOfAttribute (char*)
The *valueOfAttribute* functions return a pointer to the value (an instance of class ValueType) of the attribute specified by their arguments. The argument is either an ATTR_ENUM (the usual case) or the name of an attribute.

The intended use of this class is illustrated in the following two examples. The first shows how to create a new list of attribute/value pairs. The second shows how to retrieve values from such a list.

```
{
   ...
   /* Allocate a new FeatureList. */
   FeatureList* fList = new FeatureList;

   /* Include the pair whose attribute is ATTR_NAME
      and whose value is the symbol aName. */
   fList -> include (ATTR_NAME, VT_SYMBOL, aName);

   /* Include the pair whose attribute is ATTR_TYPE
      and whose value is the type TYPE_FIXTURE. */
```

```
    fList -> include (ATTR_TYPE, VT_TYPE, TYPE_FIXTURE);
    ...
}

{
    /* Assume that fList is the FeatureList created
       in the above example. One can retrieve its
       values as follows. */

    ValueType* vType;
    char* aSymbol;
    TYPE_ENUM aType;

    vType = valueOfAttribute (ATTR_NAME);
    aSymbol = (char*) vType;

    vType = valueOfAttribute (ATTR_TYPE);
    aType = (int) vType;
}
```

9. Class Feature List Iterator

Instances of class FeatureListIterator allow one to step through the list of pairs of a FeatureList. Each call to the member function next() returns a pointer to an AttributeValuePair. A zero is returned after all the pairs have been returned. Currently supported functions include:

FeatureListIterator (FeatureList&)
: The constructor takes a reference to a FeatureList as its sole argument.

AttributeValuePair* first ()
: The function *first* resets the iterator and returns a pointer to the first AttributeValuePair in the FeatureList.

AttributeValuePair* next ()
: The function *next* returns a pointer to the AttributeValuePair following the last one returned by this function, or to the first AttributeValuePair in the FeatureList if none has been returned from the FeatureListIterator since it was created. A zero is returned when a pointer to every AttributeValuePair has been returned.

This class is provided so that one can sequentially process all the pairs in a FeatureList without having to know the internal representation of feature lists. The intended use is illustrated in the following example:

```
{
    /* assume that aFeatureList points to a FeatureList */

    FeatureListIterator fIter (*aFeatureList);
    AttributeValuePair* avPair;

    while (avPair = fIter.next()) {
        /* process the pair pointed to by avPair */
```

```
        . . .
    }
}
```

10. Class Value Type

The class ValueType is the base class for a set of derived classes that represent the various primitive types of values that may appear in the attribute/value pairs of messages. These include addresses, dimensioned numbers, integers, material identifiers, real numbers, strings, symbols, type identifiers, and lists of other types of values.

The two most important things that can be done with these classes is to create instances of them and to extract their underlying values. The constructors and coercion operators for these classes are documented below.

10.1. Class Address

This class represents addresses of IMW subsystems, such as PL (planner), CX (cutting expert), HX (holding expert), MX (modeler), etc. If you are generating a reply to a sentence, you do not care where it came from, and just want the reply to go back to the sender, the reply function in class Sentence takes care of creating the return address.

Address (ADDR_ENUM)
int operator int ()

> Instances of this class may be coerced to an int, which will, in fact, be an ADDR_ENUM.[8] Some programs will not care where messages come from, so long as they are meaningful messages in context.

10.2. Class Dimension

This class represents dimensioned numbers.

Dimension (int, UNIT_ENUM)
Dimension (long, UNIT_ENUM)
Dimension (float, UNIT_ENUM)
Dimension (double, UNIT_ENUM)
int operator int ()
long operator long ()
double operator double ()

> The constructors take a numeric first argument of one of four types: int, long, float, or double. The second argument specifies the units. Instances of this class may be coerced to an int, whose value will be the dimension part of the dimensioned number (of type UNIT_ENUM).[9] They may also be coerced to either a long or a double, whose value will be the numeric part of the dimensioned number.

8. Refer to the file addr.h for the definition of ADDR_ENUM.
9. Refer to the file unit.h for the definition of UNIT_ENUM.

10.3. Class Integer

This class represents integers.

Integer (int)
Integer (long)
long operator long ()

> The constructor takes either an int or a long as argument; internally the value is represented as a long. Instances of the class may be coerced to a long.

10.4. Class Material

This class represents types of materials, e.g., aluminum, steel, etc.

Material (MAT_ENUM)
int operator int ()

> The constructor takes as argument a value of the enumerated type MAT_ENUM. Instances of this class may be coerced to an int, which will, in fact, be a value of type MAT_ENUM.[10]

10.5. Class Real

This class represents real numbers.

Real (float)
Real (double)
double operator double ()

> The constructor takes either a float or a double as argument; internally the value is stored as a double. Instances of this class may be coerced to a double.

10.6. Class String

This class represents character strings.

String (/* dynamically allocated */ char*)
char* operator char* ()

> The constructor takes a null terminated dynamically allocated C string as argument. Instances of this class may be coerced to a C string.

10.7. Class Symbol

This class represents names of various things.

Symbol (/* dynamically allocated */ char*)
char* operator char* ()

10. Refer to the file mat.h for the definition of MAT_ENUM.

The constructor takes a null terminated C string as argument. Instances of this class may be coerced to a C string.

10.8. Class Type

This class represents types of feature lists.

Type (TYPE_ENUM)
int operator int ()

Instances of this class may be coerced to an int, whose value will be a feature list type (of type TYPE_ENUM).[11]

10.9. Class List

This class represents a list of values.

List ()
The constructor simply creates a new empty list.

List* append (ValueType* vType)
List* insert (ValueType* vType)

The member functions *append* and *insert*, respectively, add items to the tail and head of the list.

int length ()

The member function *length* returns the number of items in the list. The user could determine this information using a ListIterator, but *length* is much more efficient and convenient.

VECTOR operator VECTOR()
List* VectorToList (VECTOR)

Lists of three real numbers are commonly used to represent vectors. The geometric modeler used by the IMW represents vectors as a struct with three fields representing real numbers (x, y, and z). The coercion operator *VECTOR()* converts a List of three numbers into this structure representation. The function *VectorToList* creates a List with three real numbers from its VECTOR parameter.

10.10. Class ListIterator

This class allows one to step sequentially through the values of a List.

ListIterator (List& aList)
ValueType* first ()
ValueType* next ()

11. Refer to the file type.h for the definition of TYPE_ENUM.

The constructor takes a reference to a List as its sole argument. The function first resets the iterator and returns a pointer to the first ValueType in the List. The function next returns a pointer to the ValueType following the last one returned by this function, or to the first ValueType in the List if none has been returned from the ListIterator since it was created. A zero is returned when a pointer to every ValueType has been returned.

11. Example

The following code is a template from which an IMW subsystem can be developed. The implementor is primarily responsible for replacing the comment

```
/* USER CODE GOES HERE */
```

under "`case INPUT`" with whatever code segment is required to implement the subsystem. This will probably be little more than a procedure invocation. If there is one-time initialization to be done, it should be placed before the BEGIN_TASK macro.

The macros BEGIN_TASK, END_TASK, EXIT_TASK, and TERMINATE are defined in macros.h. BEGIN_TASK and END_TASK create an indefinitely long loop that gets and processes the next message sent to this task. EXIT_TASK causes the loop to terminate, and TERMINATE terminates the task. Aside from knowing the general structure imposed by these macros, the user need not worry about them.

```
#include <macros.h>
#include <parser.h>
#include "expert.h"

Expert::Expert (  char* myName,
                  MessageQHead* readFrom,
                  MessageQTail* writeTo )
: /* task */(myName, DEDICATED, SIZE)
{
   Message* request;

   /* USER INITIALIZATION GOES HERE */

   BEGIN_TASK;

   request  = readFrom -> get();

   switch (request -> mOperation) {
      case INPUT:
         {
         /* USER CODE GOES HERE */
         }
         break;
      case QUIT:
         EXIT_TASK;
      default:
         {
            Message* error = new Message (ERROR,
                  "(Expert) Illegal operation");
            writeTo -> put (error);
         }
```

```
            break;
        }

        delete request;

        END_TASK;
        TERMINATE;
    }
```

If the subsystem being implemented can perform its function without intermediate communication with other subsystems, then it can be implemented as a function that takes a Sentence as input and returns a Sentence as output.

```
Sentence* job = (Sentence*)(request -> mContents);
Sentence* reply = SimpleServerSubsystem (job);
writeTo -> put (reply);
```

In most cases this is not feasible and the user's code must be designed to deal with multiple messages per job.

12. Makefile

The following is a simple makefile for creating a program based on expert.c and the Generic Expert library.

```
# INC = <the directory containing macros.h and parser.h>
# LIB = <the directory containing libtask.a and libshell.a>

CC = CC
CFLAGS = -DPRIMITIVE -c -g -I. -I$(INC)

LIBTASK = $(LIB)/libtask.a
LIBSHELL = $(LIB)/libshell.a
MATHLIB = -lm

LIBS = $(LIBSHELL) $(LIBTASK) $(MATHLIB)

OBJECTS = expert.o

.c.o:
   $(CC) $(CFLAGS) $*.c

ex: $(OBJECTS)
   $(CC) -f68881 $(OBJECTS) $(LIBS) -o ex

expert.o: expert.h $(INC)/macros.h $(INC)/parser.h
```

13. Release Notes

The following files are needed to use the generic application shell:

libtask.a	C++ Task library (you must have the version that supports software interrupts, i.e., signals)

libshell.a	Generic application library file.
expert.c	A template for the implementation of the constructor for class Expert.
expert.h	Header file with the declaration of class Expert.

14. Limitations

There is a limit on the total size of local variables associated with the constructor of a task such as Expert. The task system default limit is defined in task.h as 750. Applications which use lots of stack space can raise this limit at task creation time. This is the function of the SIZE parameter that appears in the line

```
: /* task */ (myName, DEDICATED, SIZE)
```

at the beginning of the constructor for class Expert (see the Example section). SIZE is defined in expert.h.

15. Bibliography

Shopiro 1987 Shopiro, Jonathan. "Extending the C++ Task System for Real-Time Control." Proceedings, USENIX C++ Workshop, 1987, pp. 77-94.

Stroustrup 1986 Stroustrup, Bjarne, *The C++ Programming Language*. Addison-Wesley, 1986.

Stroustrup 1987 Stroustrup, Bjarne, and Jonathan E. Shopiro. "A Set of C++ Classes for Co-routine Style Programming." Proceedings, USENIX C++ Workshop, 1987, pp. 417-439.

Center for
Integrated Manufacturing Decision Systems

Generic Environment for Lisp-based Experts

Paul Erion

March, 1990

Abstract:

This document describes the generic expert shell that supports the common elements of the Planner, Plan Manager, and Human Interface subsystems of the IMW (Intelligent Machining Workstation).

The interfaces to the internal representation of FEL sentences is described, the dialogue mechanism is explained, and issues of integrating subsystem specific code with the generic expert are discussed. Enough information is provided to enable a programmer to develop or maintain such a program.

Copyright © 1990 Carnegie Mellon University

Contact:
David Bourne
CIMDS
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-8810

Generic Environment for Lisp-Based Experts

Not all of the IMW subsystems run under the Sun UNIX Operating System. The Planner, Plan Manager, and Human Interface are Lisp-based and run on the TI Explorer. As a consequence, a Lisp-based generic expert shell is provided for these subsystems.

This document is aimed at the developer[1] of a Lisp-based IMW subsystem. Since, communication between experts takes the form of FEL sentences, explanations and descriptions are provided for functions that ease the creation and modification of FEL sentences. Lisp forms are also described that deal with dialogues, a higher level grouping of sentences.

At the end of this section, the reader will find a glossary of functions available to the writer of a Lisp-based expert. This glossary contains some functions that were not mentioned in prior sections of the text.

1. Dialogues on the Lisp-based Version of the Generic Expert

Typically, an expert exists to perform some service requested by another expert and, for every request received, a response must be generated and returned. Since the lingua franca of the IMW is FEL, both the request and the response are in the form of FEL sentences. A dialogue is simply the exchange of FEL sentences that occurs due to one expert making a request of another. For our purposes, dialogues will be viewed as coming in two versions: simple and complex.

1.1. Simple Dialogues

A simple dialogue is defined as one in which no other experts must be contacted in order to complete the dialogue. That is, if expert A makes a request of expert B, B does not need to initiate a dialogue with any other expert in order to generate a response for A (see Figure 1).

1. In this section, the writer of a Lisp-based expert is referred to as a subsystem integrator, or simply, an integrator.

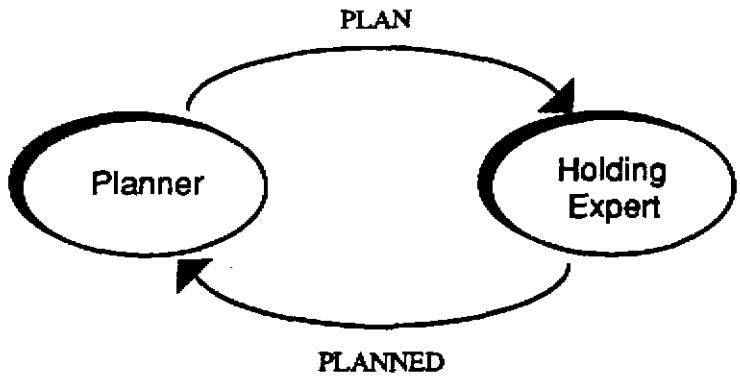


Figure 1. Example of a Simple Dialogue

1.2. Complex Dialogues

However, the system architecture of the Intelligent Machining Workstation dictates that an expert also be able to conduct complex dialogues. In other words, due to the specialization of the experts it is virtually a requirement that an expert be able to interact with other experts while performing a service triggered by an FEL sentence. A complex dialogue will be defined as one that involves separate, independent dialogues with other experts in order to generate an answer for the expert that initiated the original dialogue.

For example, the Human Interface may request that the Planner formulate a plan for a part. During the process of planning, the Planner determines that some information is required from the Holding Expert. The Planner needs to be able to acquire the information from the Holding Expert and then continue the planning process. Finally, when the planning process is complete, the result, in the form of an FEL sentence, will be returned to the Human Interface. What has transpired is two separate dialogues. A dialogue between the Human Interface and the Planner, and a dialogue between the Planner and the Holding Expert. The first dialogue, which was complex, was initiated by the Human Interface and involved the Planner. The second dialogue was originated by the Planner and involved the Holding Expert. This dialogue was a simple dialogue (see Figure 2).

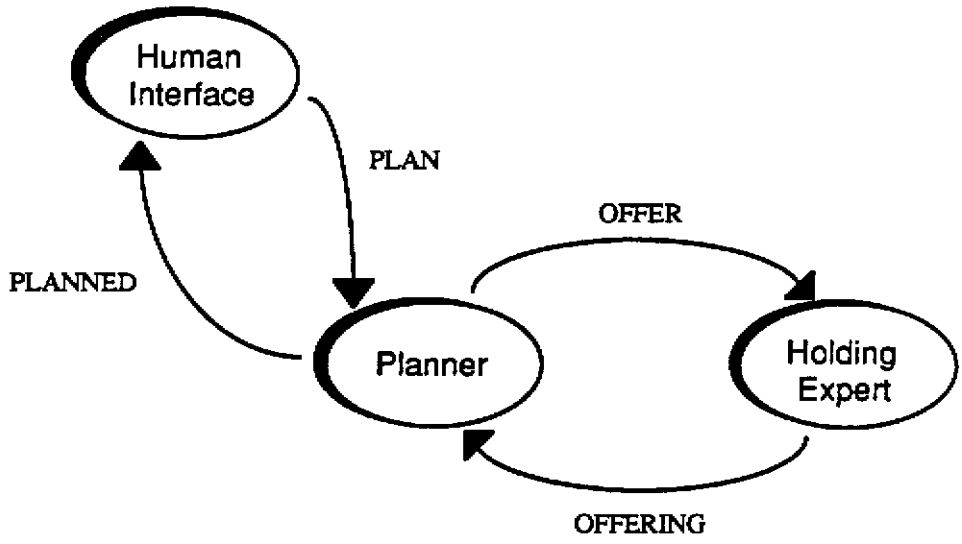


Figure 2 Example of a Complex Dialogue

1.2.1. Functions to Facilitate Complex Dialogues

The latest version of the Lisp-based generic expert provides three functions that facilitate complex dialogues:

(a) *spawn-request*,
(b) *conditional-suspend*, and
(c) *create-Qtest*.

1.2.1.1. Spawn-Request

While an expert is servicing one request, it is not at all unusual to need some additional information or a service performed by an other expert. *Spawn-request* is the mechanism provided to initiate a new dialogue. It requires four arguments. The first argument is the sentence that will initiate the new dialogue. It will be sent to the specified expert. This sentence requires a unique name, since, it is through the sentence name that the generic expert shell is able to direct incoming sentences to their appropriate destinations.

If the expert specific code uses the function bound to the *spawn* slot of an object of type *sentence*, then the integrator need not worry about creating a new sentence name. For example, if *FEL-sentence* is such an object, then evaluation by the Planner of the form:

```
(funcall (sentence-spawn FEL-sentence) :offer)
```

returns two values (a) an object of type *sentence*, and (b) the name of the sentence. The FEL form of the returned sentence is:

(offer ((name *unique-symbol*) (type message) (from pl)))

Remember the name of this new sentence is a unique symbol.

The second argument to *spawn-request* is the function used to queue sentences for output. This function is available to the subsystem integrator as the first argument passed to the expert's initial function.

Typically, a spawned dialogue will generate a result. The third argument to *spawn-request* is the disposition of this result. Currently, the third argument may be one of two symbols, either *:return-result* or *:independent*.

A disposition of *:return-result* signifies that the originating dialogue expects to have access to the final result, which should be an FEL sentence. In order to gain access to this FEL sentence, the spawned dialogue should ensure that, upon termination, this sentence is the value returned.

The symbol, *:independent*, denotes that the spawned dialogue may proceed independently of the spawner. In other words, the spawned dialogue does not in any way need to concern itself with returning a value to its parent, the originating dialogue.

The fourth and final argument to *spawn-request* is the function that will handle the spawned dialogue. The first argument will initiate a dialogue with another expert, say the Holding Expert. When the Holding Expert responds there must be a function provided to deal with the response. It may simply accept the response, or engage in a protracted negotiation with the Holding Expert. The point is that whatever occurs is determined by the provided function. This function should be written to accept the same arguments as the initial function. For example, a function that would just accept the response returned by the Holding Expert could be written as follows:

```
(defun handle-hx ( enQ-outgoing
                   Qstatus-incoming
                   deQ-incoming
                   conditional-suspend )
   "Handle the dialogue with the Holding Expert."
   (declare
      (ignore enQ-outgoing)
      (ignore Qstatus-incoming)
      (ignore conditional-suspend))

   (let ( (s  (funcall deQ-incoming)) )
      (declare
         (type sentence  s))
      (values s)))
```

1.2.1.2. Conditional-Suspend

Once a dialogue has been spawned the originator may continue processing. Of course, if the disposition given to *spawn-request* was *:return-result*, there will come a time when the response from the spawned dialogue is required for further processing. However, the originator has no way of knowing the state of the spawned dialogue. Therefore, the originating dialogue needs a way of suspending operation and waiting for the result from the spawned dialogue. This functionality is provided by *conditional-suspend*, which is available to the subsystem integrator as the fourth argument passed to the expert's initial function. *conditional-suspend* takes two arguments. The first is a test. When it evaluates to non-nil, the second argument, a function, is executed.

The test function may be supplied by *create-Qtest*, which is described in section 1.2.1.3.

conditional-suspend's second argument should dequeue the returned result from the expert's incoming queue and continue processing. It should be noted that this function is not passed any arguments when it is finally called. Consequently, the integrator should make sure that the function has access to whatever local variables or functions it may need. For example, if *continue* is the function to be called, and it expects all of the arguments that are passed into the initial function, then:

```
#'(lambda () (continue  enQ-outgoing
                        Qstatus-incoming
                        deQ-incoming
                        conditional-suspend))
```

would be the second argument passed to *conditional-suspend*.

There is a source of possible confusion that needs to be clarified. *conditional-suspend* does NOT suspend processing of the function that calls it. That is, if other Lisp forms follow this call, they are executed immediately following evaluation of the *conditional-suspend* form. What is being suspended is execution of the form that is passed as a second argument to *conditional-suspend*.

1.2.1.3. Create-Qtest

As mentioned above, *conditional-suspend* requires a form that provides a test to determine when to execute its second argument. If a dialogue is waiting for a spawned dialogue to complete, then a test is needed to make such a determination. *create-Qtest* will return just such a test. That is, *create-Qtest* creates and returns a function that will query a queue and return a boolean value depending on the outcome.

create-Qtest's primary purpose is to be used in conjunction with *conditional-suspend*. As a consequence, the following description is based upon that usage. *create-Qtest* expects two arguments. The first argument is a function that queries a queue. Of the arguments passed to the initial function, this is the second argument. *create-Qtest*'s second argument is a list of dialogue names (possibly just one). The names should be the ones for which the originating dialogue wants to wait.

1.2.2. Example

Following is an example that outlines the use of the above functions. Scenario: this is a piece of code from *PL*, the initial function of the Planner. An incoming sentence is dequeued. Some processing is performed. It is determined that a dialogue must be initiated with the Holding Expert. The request sentence is created. Once the request sentence is properly initialized, a new dialogue is spawned. At this point the Planner may continue processing. When the time comes to retrieve the response from the Holding Expert, a call to *conditional-suspend* is made.

```
(defun p1   ( enQ-outgoing
              Qstatus-incoming
              deQ-incoming
              conditional-suspend )

   (let ( (FEL-sentence  (funcall deQ-incoming)) )
      (declare
         (type sentence  FEL-sentence))

       some processing

      (multiple-value-bind (request-sentence  dialog-name)
            (funcall (sentence-spawn FEL-sentence) :offer)
         (declare
            (type sentence  request-sentence)
            (type keyword   dialog-name))

         (setf (sentence-to request-sentence) :hx)

       some processing [this would include setting up the
            sentence that will initiate the new dialogue
            (that is,the request sentence)]

         (spawn-request
            request-sentence
            enQ-outgoing
            :return-result
            #'some-function-to-handle-request)

       may do more processing (if needed)

         (funcall conditional-suspend
                     (create-Qtest
                        Qstatus-incoming
                        (list dialog-name))
                     #'(lambda ()
                           (continue  enQ-outgoing
                                      Qstatus-incoming
                                      deQ-incoming
```

```
                                                    conditional-suspend)))))
```

2. Integrating the Generic Expert and a Subsystem

The responsibility of the subsystem integrator is to provide the functionality that makes a generic expert an instantiation of an IMW subsystem. In order to accomplish this task the user needs to be able to (a) access the information contained in a request, and (b) to generate a response. Since FEL is the language of communication between IMW subsystems, both requests and responses will take the form of FEL sentences.

2.1. Accessing the Request Sentence

The generic expert shell supplies functions that extract, from the incoming FEL sentence, the information needed by a subsystem to satisfy the request. For example, similar to the C++ version of the generic expert shell, the Lisp version provides both a sentence iterator and a feature list iterator.

The access functions may be divided into two categories, those that access the top level elements of a sentence, and those that access the elements of a feature list.

2.1.1. Accessing the Top Level Elements of the Request Sentence

Following are the functions that allow access of the top level elements of a sentence. For the sake of explanation, assume that *FEL-sentence* is an object of type *sentence.*

(A) *sentence-Iterator* returns two functions that may be used to iterate through the feature lists of a sentence. The first function always returns the first feature list. Each call to the second function returns the next feature list. When the list is exhausted, *:end* is returned.

For example, the s-expression:

```
(multiple-value-bind (sIter-first  sIter-next)
      (sentence-Iterator FEL-sentence)  )
```

locally binds a function to each of the two variables: *sIter-first* and *sIter-next.* The first feature list of *FEL-sentence* is always returned by *sIter-first,* and *sIter-next* returns the subsequent feature lists.

(B) The form `(sentence-name FEL-sentence)` will return the dialogue name of *FEL-sentence.*

(C) The form `(sentence-verb FEL-sentence)` returns the verb of *FEL-sentence.*

(D) The form `(sentence-from FEL-sentence)` returns the name of the expert that originated *FEL-sentence*.

2.1.2. Accessing the Elements of a Feature List

Following are the functions that provide access to the elements of a feature list (i.e, the attribute/value pairs). For the sake of exposition, assume that *fList* is an object of type *featureList*.

(A) *featureList-Iterator* returns two functions that may be used to iterate through the feature list's attribute/value pairs. The first function always returns the first attribute/value pair of the feature list. Each call to the second function returns the next available attribute/value pair. When the list is exhausted, *:end* is returned.

For example, the s-expression:

```
(multiple-value-bind (fL_Iter-first  fL_Iter-next)
      (featureList-Iterator fList)  )
```

binds a function to each of the two local variables. *fL_Iter-first* returns the first attribute/value pair of *fList* and *fL_Iter-next* returns the next available attribute/value pair of *fList*.

(B) The form `(valueOfAttribute fList attribute-name)` will retrieve, from *fList*, the attribute/value pair whose attribute is equal to *attribute-name*.

(C) The form `(typeOf fList)` returns the feature list type of *fList*. In other words, *typeOf* returns the value of the attribute/value pair whose attribute is the symbol *TYPE*.

2.2. Creating the Response

For each request received, it is expected that the subsystem will generate a response. In order to facilitate the process, a set of functions are provided that construct and modify sentences and feature lists.

2.2.1. Creating and Modifying FEL Sentences

The most straight forward way to create a sentence that will be used as a response to a request is to evaluate the form:

```
(funcall (sentence-replyTo FEL-sentence)  symbol)
```

where,

(i) *FEL-sentence* is an object of type *sentence*, and

(ii) *symbol* is an object of type *keyword*.

FEL-sentence is the sentence that initiated the request for which the response is being prepared. *symbol* is either a verb, or the tense of a verb. If *symbol* is one of the recognized verbs, then the verb of the newly created sentence will be set to *symbol*. If *symbol* is a legitimate verb tense[2], then the predicate of the response sentence will be the verb that is the specified inflected form of *FEL-sentence*'s verb.

For example, if the value of the symbol, *FEL-sentence*, is:

```
(:offer
      ((:name :pl_1) (:type :message) (:to :hx) (:from :pl)))
```

then,

```
(:offered
      ((:name :pl_1) (:type :message) (:to :pl) (:from :hx)))
```

would be the sentence created by evaluation of either of the following s-expressions:

```
(funcall (sentence-replyTo FEL-sentence) :offered)

(funcall (sentence-replyTo FEL-sentence) :past).
```

From the preceding example it should be noted that the form:

```
(funcall (sentence-replyTo FEL-sentence) symbol)
```

does not just initialize the sentence's verb. A feature list of type *message* is also added to the response. This feature list contains (a) the name of the dialogue[3], (b) the destination of the sentence[4], and (c) the source of the sentence[5].

If, once again, the value of the symbol, *FEL-sentence*, is the request sentence, then the destination of the response sentence is equal to the result of evaluating the form, `(sentence-from FEL-sentence)`. The source of the response is set to the expert that received *FEL-sentence*. And the name of the newly created sentence and *FEL-sentence* are identical. That is, both sentences belong to the same dialogue.

A subsystem integrator is provided with the means to easily access the name, verb, source and destination of a sentence. The following functions provide that access. They all take an object of type *sentence* as an argument. Let *FEL-sentence* be such an object. The dialogue name is obtained via evaluation of the form:

```
(sentence-name FEL-sentence).
```

2. Typically, FEL provides four tenses for each verb: *present, active, past,* and *not*.
3. The name of a dialogue is the value of the attribute *name* , in a feature list of type *message*.
4. The destination of a sentence is the value of the attribute *to*, in a feature list of type *message*.
5. The source of a sentence is the value of the attribute *from*, in a feature list of type *message*.

The verb of the sentence is obtained via evaluation of the form:

```
(sentence-verb FEL-sentence).
```

The source of the sentence is obtained via evaluation of the form:

```
(sentence-from FEL-sentence).
```

And the destination of the sentence is provided by evaluation of:

```
(sentence-to FEL-sentence).
```

On rare occasions the subsystem integrator may be constructing a sentence whose verb, name, destination, and/or source are not appropriate. When this situation arises, *setf* may be used on any of the preceding four forms to alter their values. For example, to set the verb of *FEL-sentence* to *:notoffered*, the form:

```
(setf (sentence-verb FEL-sentence) :notoffered)
```

needs to be evaluated. If the name, source, or destination of a sentence is changed, then this new information is reflected in the sentence's *message* feature list.

2.2.2. Creating and Modifying Feature Lists

Once a request sentence has been created, it typically needs to filled with feature lists that comprise the response. The Lisp based generic expert provides functions to assist in the initial creation, and subsequent modification, of feature lists.

The function *create-featureList* returns an object of type *featureList*. *create-featureList* requires no arguments.

In order to add an attribute/value pair to the **front** of a feature list the expert specific code must include an s-expression of the form:

```
(funcall (featureList-include fL) attribute value)
```

where
(i) *fL* is an object of type *featureList*,
(ii) *attribute* is the name of an attribute, and
(iii) *value* is the value that is to be associated with *attribute*.

If it is desired order to add an attribute/value pair to the **back** of a feature list the expert specific code must evaluate an s-expression of the form:

```
(funcall (featureList-append fL) attribute value)
```

where
(i) *fL* is an object of type *featureList*,
(ii) *attribute* is the name of an attribute, and
(iii) *value* is the value that is to be associated with *attribute*.

A note of warning: a feature list created by expert specific code needs to contain both an attribute/value pair of type *NAME* and an attribute/value pair of type *TYPE*.

2.2.3. Adding Feature Lists to Sentences

Once the feature list is complete (that is, all of the attribute/value pairs have been added), it may be added to the response sentence. The subsystem integrator has the option of adding the feature list at the beginning of the sentence or at the end.

Adding a feature list at the **beginning** of an FEL sentence is accomplished by evaluation of an s-expression of the form:

```
(funcall (sentence-include reply) fL)
```

where
(i) *reply* is an object of type *sentence*, and
(ii) *fL* is an object of type *featureList*.

To add a feature list at the **end** of an FEL sentence, the following form must be evaluated:

```
(funcall (sentence-append reply) fL)
```

3. Symbols in an FEL Sentence

Up to this point, it has not been explicitly stated as to the package in which the symbols of an FEL sentence should reside. If all of the components of an expert were to be interned in the same package, and the current package never deviated from that package, then problems would never arise. However, this is an unreasonable assumption. For example, the application specific code for the Human Interface is read into the *CRL-USER* package while the generic expert component is read into the *GENERIC-EXPERT* package.

To alleviate any potential problems, all of the symbols in an FEL sentence must be interned in the *KEYWORD* package. This will ensure that the symbols of an FEL sentence that have the same print name will be *EQ*.

The implications for the subsystem integrator are two. First, all symbols used in the construction of an FEL sentence must reside in the *KEYWORD* package. For example, when adding the attribute/value pair *(name foo)* to the feature list, *fL*, the integrator should use the form:

```
(funcall (featureList-include fL) :name :foo).
```

Secondly, when comparing a symbol against a symbol from an FEL sentence the integrator should be cognizant of the fact that the symbol from the sentence will be interned in the *KEYWORD* package. For example, to obtain the value of the attribute *name* from the feature list, *fL*, use the form:

```
(valueOfAttribute fL :name)
```

Why use the *KEYWORD* package? The symbols of an FEL sentence are only to be used as symbolic constants. That is, the value, definition, or properties bound to the symbols are irrelevant. Common Lisp provides the *KEYWORD* package for exactly this purpose.

4. Interface Between Generic Expert and Subsystem Specific Code

To ease the process of integration, an outline of a function is provided for the user. This function, *ex*, provides an interface between the subsystem specific code and the generic expert shell. The subsystem integrator need not worry about any of the implementation details below this function.

Brief comments are interspersed throughout the function definition that follows. Some of the comments are hints as to the placement of subsystem specific code.

```
(in-package (find-package 'expert-package))

(use-package (find-package 'generic-expert))


(defun ex (enQ-outgoing
           Qstatus-incoming
           deQ-incoming
           conditional-suspend)
   (declare
      (function ex (symbol symbol symbol symbol) t))

   (let ( (FEL-sentence  (funcall deQ-incoming)) )  ; Dequeue the request.
      (declare
         (type sentence  FEL-sentence))

      (case (sentence-verb FEL-sentence)
         ;; The subsystem integrator should insert clauses for the
         ;; verbs that the expert is expected to handle.
         ;; For example, let :plan be such a verb.
         (:plan
            ;; Create iterators for the request sentence.
            (multiple-value-bind (sIter-first  sIter-next)
                  (sentence-Iterator FEL-sentence)
```

```
;; Cycle through the feature lists.
(do ( (fList (funcall sIter-first)  (funcall sIter-next)) )
    ((eq fList :end))
   (let ( (fList-type  (typeOf fList)) ; Get feature list type
          (reply       (funcall
                          (sentence-replyTo FEL-sentence)
                          :planned)) )
      (case fList-type
         ;; key-symbol is application dependent.  Normally,
         ;; there will be a clause for each feature list type
         ;; that the IMW subsystem is required to handle.
         (key-symbol

            ;; A subsystem integrator may either use the
            ;; function valueOfAttribute to obtain the value
            ;; of an attribute, or use the iterator functions,
            ;; fL_Iter-first and fL_Iter-next, to step through
            ;; the feature list.

            (multiple-value-bind (fL_Iter-first  fL_Iter-next)
                  (featureList-Iterator fList)

               (let ( (fL  (create-featureList)) )
                  (declare
                     (type featureList  fL))

                  ;; Expert dependent code should compute
                  ;; the appropriate attribute/value pairs
                  ;; and add them to fL.

                  (funcall (featureList-include fL)
                           attribute value)

                  ;; Once the feature list is complete, add
                  ;; it to the sentence to be sent as a reply.

                  (funcall (sentence-include reply) fL))))
         (otherwise
            ;; Application specific error handling.
            )))))))))
```

The following subsections provide information about miscellaneous topics that affect the writing of an subsystem's initial function (such as the example, *ex*).

4.1. Explicit Use of Packages

The Lisp-based version of the generic expert makes explicit use of packages. Normally, the symbols of a Lisp-based expert will be interned in a package specific to the expert. In the statement:

```
(in-package (find-package 'expert-package))
```

expert-package should be the package utilized by the expert.

The generic expert shell resides in the *GENERIC-EXPERT* package and exports the symbols that have meaning for the subsystem integrator. Hence, the use of the statement:

```
(use-package (find-package 'generic-expert)).
```

Evaluation of this form imports the symbols from the generic expert into the package used by the expert.

4.2. Naming the Initial Function of an Expert

When an FEL sentence is delivered to an expert, the generic expert shell determines if it is for an existing dialogue, or if the sentence initiates a new dialogue. If the sentence is to initiate a new dialogue, the generic expert calls the function provided by the subsystem integrator. *ex* was just such a function.

A convention exists for the naming of the initial function. The name of the function is the abbreviation of the expert. For example, the name of the Planner's initial function would be *PL*.

4.3. Arguments Passed to the Expert's Initial Function

Four arguments are passed by the generic expert shell to the initial function. They are as follows:

(a) *enQ-outgoing,*
(b) *Qstatus-incoming,*
(c) *deQ-incoming,* and
(d) *conditional-suspend.*

Associated with each expert are two queues. One is a queue for incoming sentences and the other is a queue for outgoing sentences (see Figure 3).

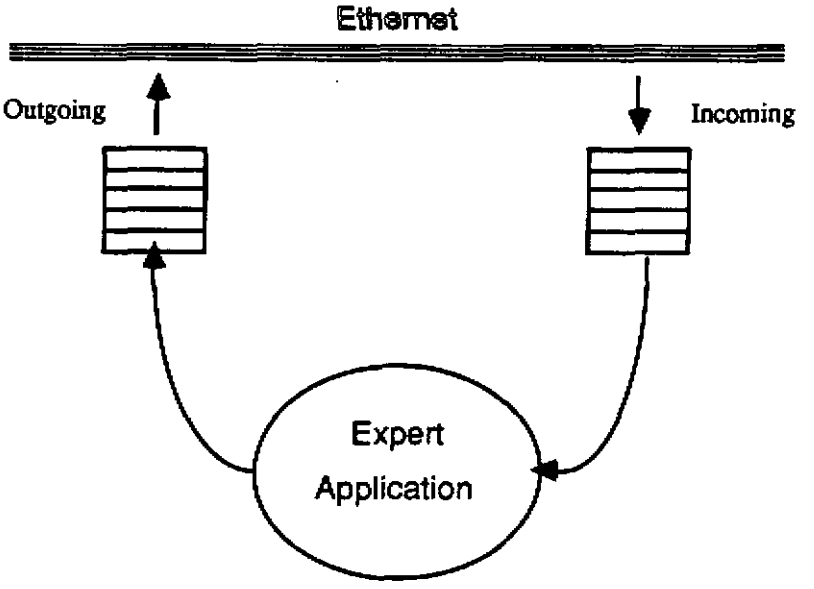


Figure 3. Incoming and Outgoing Queues

The argument *enQ-outgoing* is a variable whose value is a function that may be called to send out an FEL sentence. For example, if *reply* is an object of type sentence and a valid FEL sentence, then the form:

```
(funcall enQ-outgoing reply)
```

will place *reply* on the output queue.

Incoming sentences for an expert are placed on the appropriate queue by the generic expert shell. *Qstatus-incoming* gives access to a function that allows the integrator to programmatically check if there are any FEL sentences on the incoming queue. *Qstatus-incoming* returns *:empty* if the queue is empty, and *:non-empty* if it is not.

Of course, just knowing that there is a sentence on the queue is not much help. There needs to be a way of obtaining the sentence from the queue. *deQ-incoming* serves that function. For example, the code segment

```
(let ( (FEL-sentence  (funcall deQ-incoming)) )
   ... )
```

will pop the next entry off of the queue and bind *FEL-sentence* to it. If the queue contains no entries, then *deQ-incoming* returns *:empty*.

conditional-suspend is used when an expert needs to suspend execution until a particular event occurs. The details of *conditional-suspend* are described in section 1.2.1.2.

4.4. Branch on Verb of FEL Sentence

Typically, an expert must be able to handle more than one verb. For example, the initial function of the Plan Manager is currently able to process the verbs *:execute* and *:receive*. The example initial function, *ex*, employs a *case* statement to branch on the verb of the incoming FEL sentence. For example, following is the outline of the *case* statement used by the Plan Manager to branch on the verb of the incoming sentence, *FEL-sentence*.

```
(case (sentence-verb FEL-sentence)
   (:execute
      ... )
   (:receive
      ... )
   (otherwise
      (format *terminal-io* "~&Unexpected Verb~%")))
```

5. Creating an Instantiation of a Lisp-Based Expert

Once the user has provided the functionality required by a particular subsystem, an instantiation of that expert is accomplished by evaluation of the function *expert*. The only required argument is the name of the expert which the instantiation will represent. For example, for an instantiation of the Planner the name would be *PL*.

The generic interface provides two interfaces to an expert: (a) a network interface, and (b) a terminal interface. The user, at the time of instantiation, specifies which interfaces will be made available to the expert. This is accomplished through the use of keyword options. That is, for each of the interfaces there is a keyword option which determines if the interface will be operable during the current instantiation. The keyword option for the network interface is, *:network-medium*, and for the terminal interface, *:terminal-medium*. Either, or both, of these interfaces may be provided to the user.

6. Network Interface

The network interface is the default interface. If the network interface is not desired, then the keyword option *:network-medium* should be given the value of *nil*.

Before creating a Lisp-based IMW subsystem there is a matter concerning network communication that needs to be discussed. The network communication software currently requires a file, *Bascom.db* , that lists the experts in the IMW system and specifies over which ports they communicate. If the network communication software is to be used then this file must be properly configured (see documentation on BASCOM). As provided, the file lists two experts: *PL* and *DUMMY*. This may be changed by editing *Bascom.db*. With the current file organization, this file is located in the *Generic-ExpertCommunication.Low-Level;* directory.

7. Terminal Interface

The default action of the generic expert is also to provide a terminal interface. If the terminal interface is not desired, the keyword option *:terminal-medium* should be given a value of *nil*.

Why provide a terminal interface? An IMW subsystem may be viewed as a filter; an FEL sentence is input, some processing is performed, and then an FEL sentence is output. During development and testing, it is occasionally desirable to have total control over the input to the subsystem. Then, after the processing is complete it would be nice to easily view the output. As mentioned in the preceding section, the experts communicate with each other via the ethernet. This process of network communication implies the existence of at least two functioning IMW subsystems. On occasion, this complicates the process of development. Consequently, a terminal interface is provided that will allow an instantiation of an expert to function stand-alone. That is, the input is provided to the expert via the terminal interface and the output from the expert is displayed on the terminal window.

In order to obtain a stand-alone configuration, enter the command:

```
(generic-expert  expert-name
                 :network-medium  nil)
```

During evaluation of the above function call, the user will be required to provide the upper-left and bottom-right hand corners of the interface window. The corners are chosen via the mouse.

Initially, the interface window is not visible. When it is needed the window needs to be selected. This may be accomplished by bringing up the Explorer System Menu. In the *WINDOWS* column click on the entry named *Select*. This will bring up a menu that will list the currently available windows. One of those will have the name of the expert (e.g., *PL*), click on that name. This will bring up the interface window. At this point, FEL sentences may be input to the expert for processing. That is, the sentence may be typed into the window. After processing, the resulting FEL sentence will be displayed in the window.

Don't be misled by the above discussion of a stand-alone subsystem. It is also possible to have both a terminal interface and a network interface for an expert. In that way you can provide input to one expert and have the output sent along to another expert.

As an example, let's assume that we have two experts: a holding expert (HX) and a sensing expert (SX). If the command:

```
(OFFER
   ( (NAME SX_OFFER_1)
     (TYPE MESSAGE)
     (TO HX)
     (FROM SX)
     (MEDIUM (:NETWORK)) )
   ( (NAME SX_SELECT_1)
     (TYPE SELECT)
     (FIXTURES MAX) ))
```

is entered into the terminal interface for the HX subsystem, it will receive the FEL sentence as if the source were the SX subsystem. Therefore, the reply generated by HX will be sent over the network to SX.

In the above example, there is one detail worth noting; that is, the attribute/value pair *(MEDIUM (:NETWORK))* in the feature list of type *MESSAGE*. In the current implementation of the generic expert, the medium over which a reply will be sent defaults to the medium over which the request was received. In other words, if the request comes in over the network the reply goes out over the network and similarly for the terminal interface. The attribute *MEDIUM* provides a method for overriding the default. Hence, if a sentence is input from the terminal interface, but it is desired that the response go out over the network then that can be accomplished. It is also possible to have a response go out over both mediums. A value of *(:NETWORK :TERMINAL)* for the attribute *MEDIUM* will send the reply out over both.

8. Setting up the Generic Expert on the Explorer

The files that define the Lisp based generic expert may be obtained by tape or by copying the files from the directory *Generic-Expert* on the machine *kafka.imw.ri.cmu.edu*.

i) Copy the files into a directory named *Generic-Expert* on your machine. All is not lost if some reason exists that prevents the use of a directory by that name. The files may be loaded into a directory of your choice. However, by not using the default, some additional work is required. After the files are copied into a directory, the file *Generic-Expert.Translations* must be edited to reflect the actual location of the files. It should be noted that the *Generic-Expert.Translations* file is not required if a *Generic-Expert* entry, with the appropriate translations, is added to the network namespace.

ii) Move the files *Generic-Expert.System* and *Generic-Expert.Translations* into the *Site* directory.

iii) From a Lisp Listener, evaluate the command:

```
(make-system 'generic-expert :compile :noconfirm).
```

This will load the Lisp based generic expert skeleton into the environment.

9. Glossary of Functions for Lisp-based Generic Expert

```
(make-system 'generic-expert :compile :noconfirm)
```

Install the Lisp-based generic expert on a TI Explorer. This assumes that the source code exists on the machine in question.

```
(ge:expert 'XX)
```

Instantiate an expert. *XX* is the initial function for the expert. This function name should also be the two letter abbreviation of the expert. For example, the initial function for the Planner would be *PL*. The default for the function *ge:expert* is to provide a terminal and network interface.

The symbol *expert* is exported by the *GENERIC-EXPERT* package, so if the user executes the form:

```
(use-package (find-package 'generic-expert))
```

then access may be gained to *ge:expert* without the annoying package prefix (that is, *ge:*).

Following is the basic outline of the initial function for an expert. In this example, the expert is named *XX*.

```
(defun XX  (enQ-outgoing
            Qstatus-incoming
            deQ-incoming
            conditional-suspend)
     (declare
         (function XX (symbol symbol symbol symbol) t))

   (let ((FEL-sentence (funcall deQ-incoming)) )
       (declare
          (type sentence  FEL-sentence))

        (case (sentence-verb FEL-sentence)

             ;* Clauses for the verbs that expert XX is
             ;* expected to handle

           (otherwise
             ;*  Application specific error handling
             )))
```

```
(funcall enQ-outgoing  FEL-sentence)
```

enQ-outgoing takes one argument, an object of type *sentence*. *FEL-sentence* is placed on the queue which is read for output. *enQ-outgoing* is not a global function. It is one of the arguments passed to the expert's initial function.

`(funcall` **`Qstatus-incoming`**`)`

Returns *:non-empty* when the incoming queue has at least one unread object of type *sentence* on it. *:empty* is returned when the queue is empty. *Qstatus-incoming* is not a global function. It is one of the arguments passed to the expert's initial function.

`(funcall` **`deQ-incoming`**`)`

Returns an object of type *sentence*. FEL sentences placed on this queue are to be read as input. *deQ-incoming* is not a global function. It is one of the arguments passed to the expert's initial function.

`(funcall` **`conditional-suspend`** `test fcn)`

conditional-suspend takes two arguments. The first is a test. When it evaluates to non-nil, the second argument, a function, is executed. *conditional-suspend* is not a global function. It is one of the arguments passed to the expert's initial function.

`(`**`create-Qtest`** `Qstatus dialogNames)`

Returns a function that queries the queue whose status is obtained via the function *Qstatus*. The query is determined by the value of *dialogNames*, which should be a list of dialogue names. That is, the returned function will return non-nil if sentences exist on the queue that are associated with the names in *dialogNames*; otherwise nil is returned. Typically, *create-Qtest* is used in conjunction with *conditional-suspend*.

`(`**`spawn-request`** `FEL-sentence enQ-outgoing disposition fcn)`

This function initiates a new dialogue. It requires four arguments.

FEL-sentence is the sentence that will initiate the dialogue with the designated expert. The *spawn* slot of a *sentence* object is normally used to create this new sentence.

enQ-outgoing is the function used to queue the sentence for output. This function is an argument passed to the initial function of the expert.

disposition is the disposition of the result of the spawned dialogue. Currently, this may be one of two values, either *:return-result* or *:independent*. A disposition of *:return-result* signifies that the originating dialogue expects to have access to the final result, which should be an FEL sentence. In order to gain access to this FEL sentence, the spawned dialogue should ensure that, upon termination, this sentence is the value returned. *:independent* signifies that the spawned dialogue may proceed independently of the spawner.

fcn is the function that will handle the spawned dialogue.

(create-featureList)

Returns an object of type *featureList*.

The next three functions take an object of type *featureList* as an argument. Let *fList* be such an object.

`(featureList-iterator fList)`

Returns two functions that may be used to iterate through the attribute/value pairs of a feature list. The first function always returns the first attribute/value pair of the feature list. Each call to the second function returns the next attribute/value pair. When the list is exhausted *:end* is returned.

`(funcall (featureList-append fList) attribute value)`

Associate *value* with *attribute* and add this attribute/value pair at the end of the feature list's attribute/value pairs.

`(funcall (featureList-include fList) attribute value)`

Associate *value* with *attribute* and add this attribute/value pair at the beginning of the feature list's attribute/value pairs.

`(funcall (featureList-update fList) attribute value)`

Find the attribute/value pair in *fList* with an attribute equal to *attribute*. Change the value of this pair to *value*.

The following functions take an object of type *sentence* as an argument. Let *FEL-sentence* be such an object.

(sentence-iterator FEL-sentence)

Returns two functions that may be used to iterate through the feature lists of the sentence. The first function always returns the first feature list. Each call to the second function returns the next feature list. When the list is exhausted *:end* is returned.

It should be noted that the feature list containing the attribute/value pair `(type message)` is not returned by either of the iterators. The information that is contained in that feature list is to be obtained more directly. For example, *sentence-to* returns the destination.

(sentence-name FEL-sentence)

Returns the dialogue name of *FEL-sentence*. *setf* may be used on this form to change the value.

(sentence-verb FEL-sentence)

Returns the verb of *FEL-sentence*. *setf* may be used on this form to change the value.

(sentence-from FEL-sentence)

Returns the source of *FEL-sentence*. *setf* may be used on this form to change the value.

(sentence-to FEL-sentence)

Returns the destination of *FEL-sentence*. *setf* may be used on this form to change the value.

(sentence-medium FEL-sentence)

Returns a list whose elements are the medium of *FEL-sentence*. *setf* may be used on this form to change the value.

If *FEL-sentence* is a sentence to be output, then the value of the above form gives the media over which this sentence will be sent. For example, if evaluation of the form:

```
(sentence-medium FEL-sentence)
```

produced the result:

```
(:network :terminal);
```

then, when output, the *FEL-sentence* would be sent to both the network handler and the terminal handler. If *FEL-sentence* is a sentence that was sent to the expert, then the form, `(sentence-medium FEL-sentence)`, gives the medium over which the sentence arrived. Normally, the subsystem integrator need not worry about the input and/or output media.

`(funcall (`**`sentence-include`** `FEL-sentence) feature-list)`

Include *feature-list* in the sentence *FEL-sentence*, where *feature-list* is an object of type *featureList*.

`(funcall (`**`sentence-replyTo`** `FEL-sentence) verb-or-tense)`

Returns an object of type *sentence*. The verb of the newly created sentence is determined by the value of the symbol *verb-or-tense*. If the value of *verb-or-tense* is one of the recognized verbs, then the verb of the sentence is simply the value of *verb-or-tense*. If *verb-or-tense* is a legitimate verb tense, then the predicate of the response sentence will be the verb that is the specified inflected form of *FEL-sentence*'s verb. The destination of the sentence is equal to `(sentence-from FEL-sentence)`. The source of the sentence is set to the expert that received *FEL-sentence*. The name of the created sentence and *FEL-sentence* are identical.

`(funcall (`**`sentence-spawn`** `FEL-sentence) verb)`

Returns an object of type *sentence*. The verb of the newly created sentence has been set to *verb*. The source of the sentence is set to the expert that received *FEL-sentence*. The destination of the sentence is not set. A unique symbol is created for the name of the sentence.

The next two functions take an attribute/value pair as an argument. An attribute/value pair is a list of two elements. The first element is a symbol. The second element may be one of four types: a symbol, a number, a string, or a list. In the

examples, let *pair* be an attribute/value pair.

`(attributeOf pair)`

Returns the attribute of *pair*.

`(valueOf pair)`

Returns the value of *pair*.

These last functions take an object of type *featureList* as an argument. Let *fList* be such an object.

`(nameOf fList)`

Return the value of the attribute/value pair whose attribute is the symbol *NAME*.

`(typeOf fList)`

Return the value of the attribute/value pair whose attribute is the symbol *TYPE*.

`(valueOfAttribute fList attribute)`

Return the value of the attribute denoted by *attribute*.