

Using the Feature Exchange Language in the Next Generation Controller

David Alan Bourne
Duane T. Williams

CMU-RI-TR-90-19

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

August 1990

Copyright © 1990 Carnegie Mellon University

The work described in this document was supported by Martin Marietta, Information & Communications Systems as part of the Next Generation Workstation/Machine Controller (NGC) Program.

Contents

1. Executive Summary.	1
2. Introduction.	3
3. Current Status of FEL.	7
4. Functional Requirements for NML.	14
5. Programmability.	20
6. Graphical Interfaces to FEL and NML.	31
7. Efficiency Issues with FEL.	33
8. Enhancements to the Generic FEL Application.	35
9. Conclusion.	41
Bibliography.	43
Appendix A: Commentary on the NGC Requirements Definition Document.	44

Abstract

The Air Force has two ongoing initiatives to aid the ailing U.S. Machine Tool Industry. The first is the Intelligent Machining Workstation (IMW), which has the goal of automatically producing one-off quality parts. The second is a Next Generation Controller (NGC) initiative, which has as its primary objective to design and specify an open architecture controller for machine tools.

This report analyzes whether the integration language developed for the IMW is adequate to support the requirements of an integration language needed to build the NGC. The IMW's Feature Exchange Language (FEL) is a simple message oriented language designed to integrate diverse modules. The NGC has a specified need to design a Neutral Manufacturing Language, which can be readily used to integrate diverse third-party modules into a coherent controller. We show how with a few minor extensions FEL can be used to meet this need.

1. Executive Summary

The Feature Exchange Language (FEL) is a simple language that was designed for communicating modules. It was initially developed at Carnegie Mellon as part of the Air Force's Intelligent Machining Workstation (IMW) program. In this context, FEL was used to represent part geometry and communicate between IMW modules: a planner, a modeler, a cutting expert, a holding expert and a low level controller.

The Next Generation Controller (NGC) has some modules in common with IMW, but it also emphasizes lower level processing and timing constraints, namely, the system must be fast enough and it must act on time. In addition, the NGC must be designed so that third party vendors can readily include their modules (e.g., a new collision avoidance package or a new board for 3-axis motion control).

As part of the NGC architecture, a Neutral Manufacturing Language (NML) is presumed to tie together the various NGC modules. However, this language has also been envisioned as a means of programming the NGC, which could mean many things depending on the components of a particular controller. For example, a particular NGC controller could have a 3-axis motion controller, a set of special purpose vision boards and a higher level part programming environment. All of these components must be programmed in our new language. The question answered by this report is: "Does FEL suffice and, if not, how would it have to be extended to satisfy the requirements of NML?"

This report gives several different detailed examples of how FEL could be used to program different representative NGC modules and their connections. For example:

- (1) **Programmable Modules or Boards** - Some modules and board are programmable in the sense that they can interpret their own language complete with control structures and other programming devices. In this case, the program or parameters to a program must be represented in NML.
- (2) **Module Configuration or Board Configuration** - Some modules (or boards) are mostly hardcoded for a particular task, but it is possible to set key registers and control the path of data and a few internal functions. In this case, the names of the registers and their values must be represented in NML.
- (3) **Part Descriptions (like PDES)** - The NGC is going to include a planning module and it will almost certainly be feature oriented. In this case, the features of the part and their parameters must be represented in NML.
- (4) **Process Oriented Part Descriptions (like NC data)** - A design feature of a part does not necessarily match a process features (i.e., how that feature is made). In this case, the process data must be represented in NML.
- (5) **Connecting Modules** - There are many possible modules that can make up an NGC and they must all communicate in NML.

These examples describe what NML must represent. And yet there are other key constraints on NML, which will shape the language and its implementation.

Perhaps, the most important additional constraint is *performance*. In this case, it should be possible to pass messages between modules very quickly, for example, it would be possible to have a binary representation of FEL messages and that the only data that is actually sent is a pointer to the message. The purpose of this extra level of representation would be to eliminate overhead between modules so that third party vendors could design cooperating modules even for time critical tasks (e.g., a collision avoidance algorithm internal to a motion controller).

However, no matter, how the messages are sent they should have a uniform view to a human developer, system tester or diagnostician. Therefore, the tool used by these users of the system should be able to transparently view messages in the most readable way (e.g., a text representation or an iconic representation).

Some components of the language are task independent and some are task dependent. The task independent elements should become a standard part of the language, while the dependent parts should be possible to change or update according to the particular NGC being configured and what modules make it up. Some examples of task independent features of the language include:

- (1) **Transport information** - The name of the sender and the receiver of a message is necessary to deliver a message.
- (2) **Time information** - Because most of the information in an NGC is time critical, we are choosing to make time constraints a standard component of a message.
- (3) **Protocol information** - There are various protocols between modules and the message should reflect the state of a particular transaction (e.g., if the protocol is simply SEND and RECEIVED then a message should be marked as one or the other).

We believe that a simple language like FEL will make a good initial model for NML, because it is simple enough to convey to third party developers and it makes it feasible for programs to automatically generate and understand messages. However, it does place some constraints on the NGC architecture, namely, most of the control structure of an NGC must be internal to the modules rather than between modules.

2. Introduction

This document is a detailed, comparative evaluation of the Feature Exchange Language (FEL) and the Neutral Manufacturing Language (NML), aimed at determining the extent to which FEL already achieves the goals of NML.

2.1. Languages and Computational Models

The first question that must always be asked when designing a new language is: why? There are two simple reasons for having a language in the first place and possibly designing a new one:

- (1) **Standard Communications** - The reason to be for a language is simply a convention for communicating ideas between people, people and machines and just machines. Clearly, every participant in a communication must be able to "speak the language" in order for it to be considered successful. The problem is that in manufacturing virtually every system is programmed differently and wants to communicate in a different way. It is no wonder that true Computer Integrated Manufacturing has proved to be so difficult.
- (2) **A Good and Simple Way to Think** - Besides multi-party communications, a language also gives people and machines a way to "think." For example, there are some simple analysis problems that are virtually impossible without calculus, but with it are incredibly simple. Yes, calculus is a language, it provides new words (and therefore new concepts) and legal ways of manipulating and combining those concepts. When computers are involved, we call this a computational model.

The objectives of the NGC project have strong requirements for a standard way of communicating between many different types of computer modules, as well as providing a good and simple way to describe the solutions to all sorts of control problems.

2.2. A Brief History of Manufacturing Control Languages

This section gives a brief and incomplete history of programming languages used for manufacturing to try and pinpoint what has been the result of past attempts to introduce something new.

An early Air Force program in the 1950's also had an objective of standardizing the way we control machines. The result was Numerical Control data or NC programming, which frankly was almost too successful. It was such a radical improvement over past ways of controlling machines, that it has lasted until now and is still going strong. However, the world has not stood still during this time and our ideas and understanding of computer science and computer languages have gone through major changes, possibly 5 times. With each one of these changes, there was an attempt to keep up in manufacturing circles; however, it was only the Fortune 100 companies that could really manage these changes.

As Fortran became the dominant scientific programming language for computers in the 1960's, APT (A Programming Tool) was developed to try and keep pace by providing Fortran-like representations of numeric formulas. The problem with APT was that it required the use of the "large business machine" in order to convert it into NC data. Small companies didn't have this machine at all, and big companies had feuds over who could use the "business machine" and who had priority.

In the late 1960's and 1970's, computer science began to advance quickly as the "structure of programs" was better understood. As a result, "structured programming" became popular with languages like Algol and Pascal (and many others) gave programmers simple tools to organize and manage the control structures in their programs. These languages evolved their way into still other languages such as C and ADA, which address the need to have low level access to a machine and to manage multi-million line projects, respectively.

To some degree, manufacturing skipped structured languages altogether, although there were a few important forays into very high level languages by powerful research groups (such as IBM's AUTOPASS, but this was a project before its time). About 10 years later, small companies that made robots and vision machines, began to develop structured languages (e.g., Unimate's VAL and Automatix's RAIL). Finally in the 1980's, structured languages caught hold in the biggest manufacturing companies (e.g., GM's Karel and IBM's AML) and they also appeared in more of the new products (e.g., Adept's V+).

In the late 1980's and now the 90's, everyone is discussing object oriented programming based on Smalltalk, C++ and other languages that have been developed in computer science. Also Symbolic languages based on LISP, PROLOG and various Expert System shells have been used to some degree. However, not much of this has made its way into broadly accepted practice.

From these experiences, we can see an unusual trend. Programmers complain, but in a few years they move on to the new methods. However, in manufacturing there is a sense in which only 10% of the community moves on to the next computational paradigm. The result is that NC programming is still dominant, and the chief contender is still ladder logic that predates NC programming.

2.3. Other Areas Manufacturing has Used Languages

The previous section concentrated on languages that control machines. This section will just list a few other areas that have been accepted and used by the manufacturing community:

- (1) **Business** - The business groups in manufacturing have essentially kept pace with IBM, they have used COBOL, PL/1 and RPG to do their work without too much regard to anyone else. Finally in the 1980's, this has changed somewhat as more and more businesses rely on personal computers and electronic spreadsheets, and specialized accounting packages.

- (2) **Design** - Designers in manufacturing companies have also gone off on their own and developed various methods of describing geometry and other critical product data. As a result standards, such as IGES and tolerance standards have emerged and they are now quickly evolving to more symbolic, or feature-based, representations of products (e.g., PDES).
- (3) **Machine to Machine Protocols** - During the 1970's, manufacturing companies were hopeless when it came to protocols for tying machines together. This was realized in the 1980's; many companies participated in the development of the Manufacturing Automation Protocol (MAP), while those who could not wait started to use the computer companies' accepted products (Xerox's Ethernet, DEC's Decnet and IBM's token ring). While we have had many successful Machine Tool and AUTOFAC shows, I don't think we have any **overwhelming** success at this on the factory floor.
- (4) **Research Groups** - There has been no lack of research interest in these areas, but the impact has been relatively modest. Carnegie Mellon's CML, MIT's LAMA, NBS's hierarchical workpackages, and IBM's many efforts have probed the limits of technology.

2.4. The FELs and NMLs Place in This History

The purpose of the NGC and the trend in manufacturing is to begin the process of combining these diverse aspects of the manufacturing business and the product life cycle. For example, tool rooms, stock rooms, part programming, manufacturing engineers, machine operators, maintenance people, and system developers have historically been in different departments and used different computational tools. The NGC is starting to bring together these various concerns and therefore bringing together the tools of the past. But they don't fit together.

The Feature Exchange Language faced this problem in the Air Force's Intelligent Machining Workstation program. Industry suggested using various languages for various tasks: PDES for part description, NIST's workpackages for control, NC programs for running the machine, a standard expert system shell, MAP and others (all at once, all in the same system). Each of these choices is probably fine in isolation, but when put together they represent a design and programming nightmare. For this reason, FEL was designed to be as simple as possible, while still covering all of these different areas.

NML has the same problem as FEL, it too is trying to bring together many diverse areas of the manufacturing business. In many ways, it has to represent all of the areas that were covered by FEL, but in addition it has to reach down into the bowels of the control and be able to cope with the real time issues that face a controller. Therefore, this document will address these new constraints head on and consider whether or not FEL can be extended to fill the role of NML.

2.5. The Organization of This Report

The sections that follow this introduction were written with the follow goals in mind:

- (1) to describe the current state of FEL, both to initiate readers not already familiar with FEL and to provide a basis for evaluating the extent to which FEL already meets the requirements for NML;
- (2) to provide a conceptual framework in which to think about the requirements of NML and to say what we believe the requirements are from the point of view provided by the fundamental questions that define this framework;
- (3) to show by example how FEL satisfies many of the requirements of NML; and
- (4) to explain some enhancements to FEL which we believe will significantly improve the language and its operational environment and will remedy some of its key deficiencies and make it a reasonable choice for NML.

Section 3 (re item (1) above) describes the current status of FEL, including its syntax, semantics, and standard execution environment. Section 4 (re item (2) above) discusses requirements for NML in both general and specific terms. It includes a topic which we feel is especially important: high-level module-to-module protocols. Sections 5 through 7 aim to satisfy item (3) above. Section 5 contains a set of realistic examples of how FEL might be used to program various levels of an NGC. The examples employ control and sensing hardware with which we have first-hand experience. Section 6 discusses human interface issues. Section 7 talks about the efficiency of FEL. Section 8 (re item (4) above) describes some enhancements that we would like to make to FEL and its runtime environment, and the motivation for them. Included are: real-time support, better dialog support, simplified extensibility features, and improved error handling. Section 9 contains our general conclusions. An appendix includes the RDD items pertaining explicitly to NML and a few comments on each of them.

3. Current Status of FEL

The current status of FEL, as an operational component of the our IMW prototype modules, is explained in considerable detail in our report *The Operational Feature Exchange Language* (see [1]). In this section we summarize, for those who have not read the previous report, the main features of the language and of the generic module application in which it is embedded.

3.1. Basic Syntax

3.1.1. The Grammar

The design of FEL syntax is based on a few simple concepts: (1) sentences, (2) verbs, (3) attributes with associated values, i.e., attribute-value pairs, and (4) lists.

An FEL sentence consists of a verb and one or more lists of attribute-value pairs. An FEL verb is a symbol from a table of legal verbs. An FEL attribute is a symbol from a table of legal attributes. An FEL value is an integer, a real number, a dimensioned integer, a dimensioned real number, a symbol, a string, or a list of values. We plan to extend the syntax to effectively allow a list of attribute-value pairs to be the value of an attribute. This new type would be similar to a *struct* in C and a *record* in Pascal. (Currently, if such a list were to appear in the value position of an attribute-value pair, it would be treated as a list of lists of uninterpreted symbols and other values. The mechanisms that find and extract the values of attributes do not work on such lists.)

Here is what a partial grammar for FEL looks like:

Sentence	=	"(" <Verb> <List of Feature List> ")" "(" ")"
List of Feature List	=	<Feature List> <List of Feature List> NIL
Feature List	=	"(" <List of Pairs> ")"
List of Pairs	=	"(" <Attribute> <Value> ")" <List of Pairs> NIL

The following is an example of a typical FEL sentence, in this case a portion of the response of the IMW Process Planner to a request to plan setups for a part to be machined:

```
(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)) )

...
)

```

Code Example 1

This entire example is a single FEL sentence with an initial verb, PLANNED, followed by several feature lists. Each feature list has the required TYPE and NAME attributes, which together identify the role that list plays in the context of the sentence. For example, the last feature list has TYPE 'face', indicating a face milling operation, and NAME 'face_50', an arbitrary identifier, which distinguishes this face milling from others. The other attributes that appear in the feature lists are context dependent data that describe the as-

sociated operations. For example, the last feature list has a FACE attribute which identifies the side of the stock to be milled; and the various vectors describe the volume of material that is to be removed from the stock.

In this particular example, the order of the feature lists is significant, except for the first one, with TYPE 'message', which may appear anywhere. In general, though, neither the order of the feature lists nor the order of the attribute-value pairs within feature lists matters.

3.1.2. The Parser and Sentence Generation

The details of the grammar of FEL are implemented by a parser that is automatically generated from a formal description of the language using the UNIX utilities LEX, YACC and SED. The parser converts the ASCII representation of FEL into a binary representation that can be accessed by the module.

Sentences of FEL are constructed by means of a set of parameterized functions which guarantee that the resulting binary representation is in the proper format. Moreover, the module programmer is insulated from the details of the binary format, both for construction and subsequent access, which makes the implementation of the language more robust and maintainable.

3.2. Semantics

3.2.1. Meaning is Determined by Use

The meanings of FEL sentences, verbs, and attributes are not predefined, but are, instead, determined by their use in communication between the various modules. Different pairs of modules may use the same sentences, verbs, and attributes in different ways and, thereby, assign them different meanings. In practice, of course, the differing uses of a verb or attribute tend to be related in a sensible way, and it makes sense to encourage this.

The IMW Process Planner may be asked to PLAN a part and the Holding Expert may be asked to PLAN a setup. Both uses of the verb PLAN indicate that the module receiving the request is to produce a sequence of steps for carrying out a certain action. In the one case the steps are a sequence of setups that will result in the part being produced. In the other the steps are a sequence of fixturing instructions that will result in the part being securely held during a setup. There is no reason to have two separate verbs, since each module is specialized for a particular task and there is no local ambiguity about what a verb means to a module.

This reuse of keywords in different contexts to mean different, but similar, things is considered a design feature of FEL. It helps to simplify the language by reducing the vocabulary and, thereby, making the language easier to learn.

3.2.2. High-Level Protocol Conventions

Above the byte transport protocols, FEL supports its own protocols for designating which module is to receive a message and for grouping related messages.

Every FEL sentence is required to contain a feature list whose TYPE is 'message' which contains values for the attributes TO and FROM. The value of the TO attribute, a symbolic name, is particularly important, because it is used by the intermodule communication facilities of the generic application to route the sentence to the appropriate destination. The value of the from attribute is normally used as the destination for replies.

Groups of related messages, "dialogues" (as they were called in the IMW), enable modules to track the progress of requests and to coordinate their actions. FEL support for dialogues is provided by a feature list (with "type message") that is required (by convention) in all messages and by the use of conjugated verb forms. See Code Example 2, page 11.

Every feature list in an FEL sentence is required to contain at least two attributes: TYPE and NAME. In the context of a verb, these attributes describe the role of the feature list. FEL sentences all contain a feature list whose TYPE is "message". The NAME attribute of this feature list is taken to be the name of the dialogue of which the message is a part. This allows the message to be given special significance by virtue of its playing a role in the dialogue. The role it plays is largely determined by the form of the verb.

Every FEL sentence has a tensed verb that is a conjugated variant of a root form. The variants are (1) present, (2) present participle, (3) past, and (4) a negative form that indicates failure to satisfy a request or command. These verb forms enable a module to track its progress in a dialogue. The negative form of a verb provides support for handling errors and certain kinds of errors can be handled automatically by the generic FEL application, which is described below.


```

(get
  ( (type message) (name PL_003)
    (to MX) (from PL)
  )
  ( (type object)
    (name part_123)
    (application current_part)
    (environment part_geometry)
  )
)

(getting
  ( (type message) (name PL_003)
    (to PL) (from MX)
  )
  ( (type object)
    (name part_123)
    (application current_part)
    (environment part_geometry)
  )
)

(got
  ( (type message) (name PL_003)
    (to PL) (from MX)
  )
  ( (type object)
    (name part_123)
    (application current_part)
    (environment part_geometry)
    (feature (envelope blind_hole_1))
  )
)

```

Code Example 2

Code Example 2 shows what a simple dialog looks like. In this case a process planner (PL) requests information from a modeler (MX) about an object named "part_123" that is stored in a certain area of the modeler's hierarchical database, identified by the attributes APPLICATION and ENVIRONMENT. The protocol handler in the receiving module (MX) uses the feature list with "type message" to direct the request to an appropriate task within the module, and returns to the sending module (PL) a message (with verb "getting") indicating that the request is being handled. The planner module recognizes the "getting" sentence as a response to its request because "getting" sentences are always responses to "get" requests and the sentence it receives has a "type message" feature list whose name attribute matches that of a request that it generated. The protocol handler in the planner makes sure that the "getting" sentence gets to the associated task. The modeler task interprets the remaining feature lists of the "get" sentence according to the verb. This results in the modeler retrieving the object named "part_123" and building a reply sentence (with the verb "got") naming the features that compose that object. When the planner receives this reply it will have sufficient information to request more details about the features ("envelope" and "blind_hole_1"), if it needs them.

3.3. Generic FEL Application

Within the context of the IMW, FEL is embedded within a standard application for modules (see Figure 1, page 12). This enables modules to share a common FEL parser and sentence generation facilities. The lower-level interface to the intermodule communication path is also hidden from the module-specific code provided by the module developer.

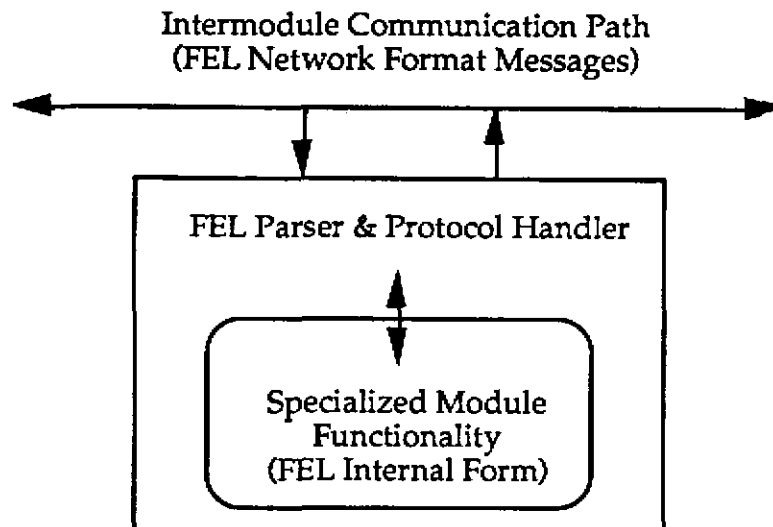


Figure 1. Generic FEL Architecture

3.3.1. Module Roles

Modules built within the generic FEL application may adopt either or both of two functional roles. A module may behave as a server process, simply responding to requests as they are received. Alternatively, a module may behave as a generator of requests in the process of carrying out a plan. The generic FEL application supports both of these roles.

The generic application allows a module to remain idle indefinitely until activated by an event, such as an incoming request. The module may then act as a server by sending a reply to the requesting module. The support provided by the generic application in this situation includes:

- (1) automatic conversion of the incoming message to an internal binary form which the module can access without knowing the format;
- (2) automatic generation of a template reply sentence, addressed to the sender of the request, into which the module can insert the context dependent details of the response;

- (3) a set of functions for constructing lists of attribute-value pairs and including them in the reply sentence without having to know the details of the sentence format; and
- (4) automatic conversion of the outgoing message to a processor neutral network format.

The generic application also allows a module to be actively engaged in a task and to also be available to respond to incoming messages, but support for this is weak. Modules have to be carefully designed to periodically check their message queues. In order to do this without risk of losing the processor for a long time, they have to arrange for an event to occur that will wake them up in case the queues are empty. The generic application could be enhanced to provide periodic wake up calls for non-server modules.

3.3.2. FEL Parser

The FEL parser is an integral component of the generic application, and its function, translating the network representation of FEL (which is currently an ASCII representation) into the internal binary form, is entirely hidden from the developer's module-specific code. This design would enable a more efficient message transfer to be implemented on a shared memory machine with no visible change to the modules.

3.3.3. FEL Sentence Generation

FEL sentences are constructed at the module level by means of functions that construct the smallest meaningful components, i.e., attribute-value pairs, and combine them as desired to form feature lists and sentences. This method of sentence generation explicitly, and deliberately, rejects the idea of hard-coding sentences, with a few parameters to be filled in at run-time, and enables the development of modules with considerable flexibility in the relation between their internal data and the resulting FEL sentence.

From the developer's point of view, this design insures that structurally invalid sentences will not be produced and possibly transmitted to other modules, while allowing an unlimited number of structurally distinct sentences to be generated. This approach eliminates much of the need for validation of individual modules in order to protect the integrity of a system.

This design also enables the underlying internal representation of sentences to be changed without affecting module-specific code.

4. Functional Requirements for NML

In this section, we look at the requirements that are imposed on NML by virtue of the role it is intended to play in the NGC. In a sense, these are the only *absolute* requirements. So, the first and most important question that we have to answer is "What is the role of NML in the NGC?"

It is possible that there are other reasonable requirements that should be imposed on NML, even though they may not be *essential* to achieving the desired functionality. For example, it is desirable that NML be relatively simple, so that it can be easily learned, quickly processed, and relatively robust. So, the second question that we will address is "What are some of the desirable general features that NML might possess?"

In subsequent sections of this report, we will explore the degree to which FEL satisfies, or might be modified to satisfy, these requirements.

4.1. What is the role of NML in the NGC?

NML has potentially two roles in the NGC. The most important is that it is to be the linguistic medium for transmitting information between modules. A secondary role is that it is to be a language in which the behavior of a module may be programmed. We will see as we develop these ideas that there is no sharp distinction to be drawn between these two roles.

4.2. Language Requirements

There are several different kinds of requirements that we can describe and analyze for the NGC and its NML language.

4.2.1. Complexity Requirements

Computer scientists study the complexity of various languages and try to understand their abstract properties. As always, we have a situation where we desire a programming language complex enough to do what is required and no more than necessary.

Complexity is usually couched in terms of the Chomsky hierarchy, which describes four general levels of languages.

- (1) Type 3 languages also known as regular languages can be represented as a finite state machine.
- (2) Type 2 languages also known as context free languages can be represented with a push down stack with finite state control.
- (3) Type 1 languages also known as context sensitive languages can be represented with 2 bounded push down stacks with finite state control.
- (4) Type 0 languages also known as unrestricted languages can be represented with 2 push down stacks and finite state control (alias the Turing

machine).

There are good reasons for choosing from the simpler end of the hierarchy, namely that the processing of the languages at each higher level is provably more complex (computationally expensive) than the previous levels, and in practice these languages are harder to implement. On the other hand, if the language of choice is too simple, it cannot be used to express some ideas that we might want to compute.

In the case of NML, we plan on operating in an environment where there are rich modules that perform complex operations (e.g., motion control, database management and planning). Therefore, we see the function of NML as the glue, which binds these subsystems together. Since the applications of NML are so varied, we must assume that the messages sent between modules should be able to carry state information, that is, the state of the communication, which FEL calls "dialogs." Furthermore, it may be possible for two modules to be communicating with a third "serving" module at the same time, which means that the state of two different communications must be maintained by the server. This complexity is necessary if *any* NML module is ever to handle multi-part messages with more than one party.

As we move up the hierarchy, the added complexity becomes more and more difficult to justify. A type two language is required to handle tasks such as balancing parentheses. In the communication domain, this would mean the ability of keeping track of dialogs within dialogs and still be able to pop your way out of a nested communication in an orderly fashion. It should be noted that considerably more mechanism is needed for nested dialogs than just handling multiple, but separate dialogs.

FEL, fully implemented to support dialogs, is a type 3 language since the complexity of the dialog transition tables have been proposed to be finite state machines. However, it could be argued that a type 2 language will be ultimately required since it may be able to handle a kind of "interrupt dialog." We've all had person-to-person dialogs where the topic of discussion is changed midstream. It is the rare occasion that people can succeed at backing their way out of a nested dialog by properly finishing previous discussions. Therefore, we do not believe that this complexity is necessary for NML.

Type 1 and type 0 languages fill out the hierarchy, and there are a few nice models for handling these richest of languages, such as, Augmented Transition Networks (ATNs), which look a little like finite state machines (e.g., states and arcs), but here the arcs can be attached to full scale programs. Again, we may eventually be interested in this additional complexity, but we will have to wait for the future computers to offer us considerably more power than is available to an NGC module.

4.2.2. Simplicity Requirement

The simplicity of a language is, on one interpretation, the reverse of complexity. But as we go up the Chomsky type hierarchy, or even as we just add more features to a language, the language becomes more and more difficult to implement and draws more and more computer resources to process it.

- (1) One major goal of the NGC and the SOSAS architecture is to make it possible for **third party developers** to make NML compliant modules. This requires a profound simplicity, since the small companies making

these products most likely have a limited talent pool for these tasks. Furthermore, the best talent in a small company is almost never placed on "integration duty."

- (2) The time constraints imposed by a real-time controller demand that the messages being sent in the NGC can be processed with a minimum of computational resources. This constraint may eventually be relaxed as we look forward to the machines of the next century, but for the decade, the time requirements of the NGC manufacturing processes will continue to stress affordable computational engines.

We believe that NML's first design should err on the side of simplicity. Once there is strong industrial acceptance, and relaxed computational constraints, it will be relatively simple to upgrade the expressive power of NML.

4.2.3. Expressivity Requirements

There has been much discussion of whether NML has to be a "full blown" programming language. Once we have chosen, languages from the bottom of the hierarchy, we must place more burden on the modules that communicate. However, we must still be able to control the function of these modules. Chapter 5 illustrates how several different kinds of modules can be controlled with a type 3 language such as FEL. Furthermore, the messages in the language should be meaningful and easy to understand. This was the reason FEL chose human readable strings to express commands with attribute-value pairs as arguments. Chapter 6 develops more fully how the human interface can be implemented and used.

Note that each statement of FEL is essentially a function call, and the power of the language is found in the organization of the messages, that is, messages taken together describe (and coerce) the flow through the appropriate finite state machine in each dialog participant.

4.3. Protocol Programming

There are two distinct styles of programming that one can envision for NGC modules. The first is "operations programming" which is a style of programming that controls the internal functions of a module. Traditional NC programming is a good example of this, since the program is loaded directly into the control box and then is interpreted to control the sequence of operations.

A second distinct style of programming is protocol programming, which controls the orderly interactions between modules. We use the term *dialog* to describe a collection of related messages that are sent between modules. As we see in the next section, protocol programming or dialogs can be used as a tool in distributed problem solving.

4.4. Is There a Need for Negotiation between Modules

There are two broad philosophies of control system design that strongly determine how systems are built and how, in particular, controllers are built. These two philosophies come under the heading of open-loop and closed-loop control.

In open loop systems, there is a built in assumption that the fundamental algorithms and/or physical process is repeatable under reasonable environmental fluctuations. For example, NC programming of a 3-axis machining process works without feedback even when stock supplies change. Minor variations in material hardness and grain direction have little or no effect on the machining process. It is said that this level of repeatability is really all that is needed in the work-a-day environment of industry.

In closed loop systems, there is a built in assumption that the fundamental algorithms and/or physical processes can never be built with all of the appropriate contingencies taken into account. To compensate for this lack of knowledge, engineers design systems that measure the environment during the process and feed these results back to the control algorithm, which then adjusts its control parameters to make up for the perceived errors. For example, some systems adjust the feed of a machining process according to an indirect measure of the cutting force. A simple instance of this is to increase the feedrate of a machining process when it is out-of-the-cut and decrease it to a specified rate as the work engages.

Oddly, each of these two points of view have strong advocates in practice. While in theory virtually no one would argue that open loop systems are better than closed loop systems, there are many designers that design systems as open loop with a stated view that feedback can be added as needed, but they then never get around to adding it. As a result, the U.S. machine tool industry builds machines that behave poorly as tools wear and as tool and part materials are changed (e.g., stock steel to stainless). In at least some of these cases, the technology and control theory are well within the state-of-the-art and yet there continues to be a strong adherence to this practice.

A closed loop system is a kind of "negotiation" between the assumptions of a hand built control module and the physical environment. Scientists are slowly realizing that this same level of uncertainty can exist between software modules, because they are often so large and complex that even their designers are no longer sure of what assumptions they have made in its construction. When two or more modules are brought together in a complex decision making situation, it is very difficult for one module to know what information the other module(s) need and how they will behave once they get it. For example, most 3D modeling systems support (unqualified) the intersection of two well defined solids. And yet, when a solid of modest complexity is: (1) copied to another object (2) rotated slightly and (3) intersected with the original object, it will result in a hard-crash of many of these systems.

The idea of "negotiation" between modules has the same underlying spirit as closed loop control, but it operates between software objects instead of a control system and its physical counterpart. The feedback provided between two modules can provide virtually any kind of information:

- (1) error messages indicating the lack of ability to carry out a particular function,
- (2) suggestions of how something might have been done better, such as the choice of a different tool or fixture,
- (3) a proposed solution to a small aspect of a complex decision problem such as process planning,
- (4) status reports, such as a task is late meeting a deadline but it will produce a good result, a task completed, or even an old fashioned error-control signal.

Just as with closed loop control systems, negotiating modules must be designed to be convergent and stable. That is, they must work together to quickly come to a solution and the suggestions (or whatever) should vary smoothly or at least consistently. For example, all closed-loop control systems oscillate (e.g., go slow, faster, faster, faster, slower...) but they should be designed to not oscillate wildly (e.g., go fast, stop, go fast, stop, ...). Both divergent and unstable behavior should be carefully avoided in the design of cooperating modules.

4.4.1. FEL's Contribution to Negotiation

While a message-oriented language such as FEL cannot guarantee the modules will behave properly, the language and its environment can make it relatively easy to build closed-loop module to module systems. FEL has been designed with this in mind.

4.5. Embedded Languages

One of the desired features of the NGC is that it should be able to incorporate existing technology. But existing devices and software systems do not understand NML. In order to use them, while satisfying the requirement that all modules communicate using NML, it will be necessary to adopt one of two approaches:

- (1) either build a sophisticated NML frontend for such devices
- (2) or build into NML the capability to transmit "foreign" languages to a relatively simple frontend that would then extract the "foreign" text and pass it directly to the device.

The latter solution has serious disadvantages. First of all, each module that needs to communicate with the device would have to contain some "knowledge" of the archaic language, even if only in a fixed collection of pre-coded programs, stored statically. This would make it more difficult to eventually eliminate the archaic language, since there would be more existing instances of it to be eliminated. The second disadvantage of solution (2) is that the burden of supporting such a device is placed on multiple users, i.e., the developers of other modules that communicate with the device, rather than on the single developer of the pre-NGC device itself. This would likely increase the overall effort (and cost) required to support the device. Aside from the initial development cost, any upgrades to the device that affect the interface may require a change

to every module that uses the device. Finally, developers, or programmers, of modules would have to learn multiple languages, rather than simply NML. For these reasons, it is not a good idea to embed other languages in NML for the purpose of supporting pre-NGC devices.

Fortunately, pre-NGC devices can be accommodated in the NGC architecture by building a sophisticated NML frontend for each of them (i.e., solution (1)). This is a task that need be done only once per device. The end result would be that the device would be instructed to perform its functions by means of standard NML expressions—such as would be used by modules originally designed to speak NML—which the frontend would convert to the (now) hidden archaic language.

It is important to realize that adopting solution (1) does not require that NML contain programming control and data representation syntax comparable to the syntax understood by all the various devices that might be retrofitted with a frontend for use in the NGC. The most that is required is that the functionality of such devices be accessible via NML. It is not necessary that that functionality be achieved in the same way that it would ordinarily have been achieved using only the archaic language built into the device. Furthermore, some of the underlying functionality may simply not be needed in the context of the NGC.

5. Programmability

In this section, the range of FEL programmability of various aspects of the NGC is illustrated by means of semi-realistic examples drawn from our experience with the IMW. We divide our examples into two broad categories: operations programming and protocol programming, with the bulk of the examples falling in the former.

Keep in mind that all of the examples shown here are just one way of employing FEL to perform the illustrated tasks. In every case there are reasonable, possibly better, alternatives that could have been chosen.

5.1. Operations Programming

Operations programming is the production of instructions that activate the internal operations of a module. The examples below illustrate the capabilities of operations programming in FEL that we believe indicate its suitability for the complex open architecture environment of the NGC.

5.1.1. Feature-Oriented Part Programming

FEL has been extensively used for feature-oriented part programming within the context of the IMW. In particular, the detailed geometrical part descriptions are introduced into the IMW in FEL format and all of the IMW test parts produced by the IMW were described in FEL.

The following example of a description of an IMW test part illustrates the how FEL can be used to convey a geometrical part description to the IMW Modeler.

```
; IMW Test Part 2

(add

; Define the object with default environment and application
; Specify position vector and three orthogonal edge vectors

  ( (type      object)
    (name      myobj)
    (p_vector  (0. 0. 0.))      ; location of one corner
    (w_vector  (2.312 0. 0.))   ; x (horizontal) vector
    (l_vector  (0. 2.582 0.))   ; y (vertical) vector
    (d_vector  (0. 0. .15)))    ; z (depth) vector

; Define the three thru holes for the object, specifying a position
; vector, the axis of the hole and its radius

  ( (type      thru_hole)
    (name      hole1)
    (object    myobj)
    (p_vector  (.375 2.207 0.)) ; location of center
```

```

        (d_vector (0. 0. .15))      ; depth vector
        (radius   .172))           ; hole radius

    ( (type      thru_hole)
      (name      hole2)
      (object    myobj)
      (p_vector  (1.375 1.312 0.))
      (d_vector  (0. 0. .15))
      (radius    .172)              )

    ( (type      thru_hole)
      (name      hole3)
      (object    myobj)
      (p_vector  (2. .312 0.))
      (d_vector  (0. 0. .15))
      (radius    .172)              )

; Define the angled side of the part as a thru slot, which is
; basically just a parallelopiped

    ( (type      thru_slot)
      (name      angle1)
      (object    myobj)
      (p_vector  (2.312 .312 0.))  ; location of one corner
      (w_vector  (1. 0. 0.))        ; x vector
      (l_vector  (-.507 2.582 0.))  ; y vector
      (d_vector  (0. 0. .15)))      ; z vector

; Define the ends of the part as swept cylinders

    ( (type      swept_radius)
      (name      topradius)
      (object    myobj)
      (p_vector  (.375 2.207 0.))   ; center of rotation
      (d_vector  (0. 0. 0.15))      ; axis of cylinder
      (w_vector  (0. 0.625 0.))     ; location of cylinder
      (angle     90.)               ; degrees of rotation
      (radius    .25))              ; radius of cylinder

    ( (type      swept_radius)
      (name      bottomradius)
      (object    myobj)
      (p_vector  (2. .312 0.))
      (d_vector  (0. 0. 0.15))
      (w_vector  (0.562 0. 0.))
      (angle     -180.)
      (radius    .25)              )

; Define the circular edge of the part as a thru hole

    ( (type      thru_hole)
      (name      cutout)
      (object    myobj)
      (p_vector  (-2.081 -1.543 0.))
      (d_vector  (0. 0. .15))

```

```

        (radius      4.235)
    )
    ( (type      message)
      (name      mymessage)
      (to        mx)
      (from      ui)
    )
)

```

Code Example 3

5.1.2. NC Programming

Traditional NC programs can be represented in FEL in several ways. A straightforward representation is shown by the following example (a program fragment) of NC code and then equivalent FEL instructions, in which each line of NC code is represented by a feature list in an FEL sentence.

```

O0030  T21121001 M06
N0040  S1528 M03
N0050  G00 X0.6000 Y-1.0000 Z2.6000
N0060  G00 X0.6000 Y-1.0000 Z2.2800
N0070  G01 X-4.1000 Y-1.0000 Z2.2800 F18.3000

```

Code Example 4

```

(execute  ( (type      ToolChange)
            (name      CX_0001)
            (toolnumber 21121001) )
  ( (type      Spindle_ON_CW)
    (name      CX_0002)
    (speed     1528 RPM) )
  ( (type      RapidPositioning)
    (name      CX_0003)
    (position   (0.6 -1.0 2.6) )
  ( (type      RapidPositioning)
    (name      CX_0004)
    (position   (0.6 -1.0 2.28) )
  ( (type      LinearInterpolation)
    (name      CX_0005)
    (position   (-4.1 -1.0 2.28)
    (feedrate   18.3 IPM) )
)

```

Code Example 5

One question that arises in an example like the above is "Why is a NAME attribute required in each of the feature lists?" Unlike the modeler example of Code Example 3, page 20, the names in this example do not seem to be playing an identifying role. But they would play such a role in the event of an error, since the names could then be used to refer to specific instructions that caused the error.

5.1.3. Motion Control

The following example illustrates how FEL can be used to send commands to a low-level motion controller (in this case the GALIL 3-axis DMC-230).¹ The motion to be produced is shown in Code Example 6 on page 23.

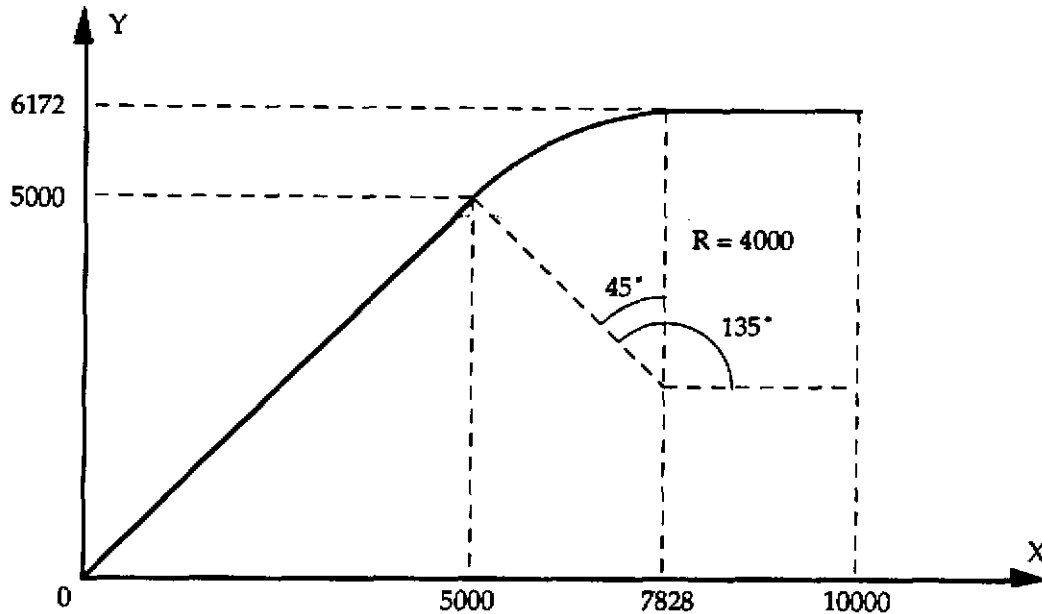


Figure 2. Motion Path for Code Example 6

One set of instructions that may be sent to the controller to produce this motion are shown in Code Example 6 on page 23 (the comments on the right in the example are not sent).

VP	5000,5000	Initial 45° straight segment
CR	4000,135,-45	Circular segment
VP	10000,6172	Final straight segment
VS	4000	Vector speed
VA	100000	Vector acceleration
BGS		Start the sequence

Code Example 6

The FEL sentences that cause this set of instructions to be sent to the controller might look like those in Code Example 7 on page 24. An important thing to keep in mind about this example is that the details of the FEL sentences need not correspond one-to-one with the details of the low-level instructions actually sent to the controller. The module that converts the FEL sentences into the low-level instructions understood by

1. The diagram and GALIL instructions for this example were borrowed from the GALIL DMC230 Series User Manual.

the hardware may be designed to provide a different, perhaps enhanced, interface to the hardware than the hardware manufacturer supports.

```
(define
  ( (type coordinated_sequence)
    (name sequence_1)
    (vector_position (5000 5000))
    (circle (4000 135 -45))
    (vector_position (10000 6172))
    (vector_speed 4000)
    (vector_acceleration 100000)
  )
)

(start
  ( (type coordinated_sequence)
    (name sequence_1)
  )
)
```

Code Example 7

For example, an alternative FEL sequence to perform the same motion is shown in Code Example 8 on page 24. Here the user defines parameters for a higher level motion, a circular corner, not directly supported by the hardware.

```
(define
  ( (type coordinated_sequence)
    (name corner_1)
    (circular_corner (
      (approach (7071.07 45))
      (turn (4000 -45))
      (follow_through (2172 0)))
    )
    (speed 4000)
    (acceleration 100000)
  )
)

(start
  ( (type circular_corner)
    (name corner_1)
  )
)
```

Code Example 8

The above example is a simple one, because the control it illustrates can be accomplished with a sequential list of parameterized commands. FEL is well suited to this style of programming, but the motion controller used in this example is capable of being more generally programmed and users will want to take advantage of the fea-

tures that this provides. In particular, the language built into the controller has variables, labels, conditional branching, and parameterless subroutines. In order to take advantage of these features, FEL would have to be significantly enhanced, or else the module that controlled the GALIL board would have to be designed with a higher level interface.

5.1.4. Image Processing

In this section, we will describe how FEL can be used to send commands to specialized image processing hardware to control real-time image processing functions of a Sensing Expert module. We will use Datacube's MaxVideo hardware and MaxWare software as the basis for our examples. These VMEbus boards, Unix device drivers, and C library interfaces are among the most sophisticated, affordable, general purpose image processing hardware available.

5.1.4.1. Datacube Interface

The standard user program for a set of Datacube boards is much more complex than the interface to the GALIL motion control described in the previous section ("Motion Control"). Datacube's MaxVideo boards are designed to be used in groups, with the boards linked together via their own private high-speed bus, in addition to the VMEbus. There are many different kinds of Datacube boards, each capable of several related functions, and many possible linkages between them. To function properly as an image processing pipeline, the boards usually have to operate in synchrony with each other. To collect meaningful data from a pipeline, the user has to wait until the end of a processing cycle. These timing dependencies are coordinated by means of interrupts which are handled and hidden from the user by a special Unix device driver. Commands to the Datacube hardware are queued and subsequently sent to the hardware after the first interrupt following a flush instruction. Special care must be taken to insure that queued and immediate instructions are executed in the desired order. In addition, there can be multiple queues whose instructions are interleaved in order to achieve real time operation. Not surprisingly, the standard programming environment uses the C programming language supplemented by libraries of Datacube supplied subroutines.

5.1.4.2. Building an FEL Datacube Interface

Despite the complexity involved in programming a set of Datacube boards, the FEL interface to a module that controls the boards can be relatively simple, because the lowest level procedures that access the hardware just set or get values in registers on the boards. This allows a straightforward FEL interface to be developed, with FEL verbs and attributes that match the Datacube procedures and their associated parameters.

The following examples illustrate this for the DIGIMAX Digitizer and Display Module's board setup and lookup table (LUT) control functions. In each case, the C interface to the Datacube hardware is shown, followed by its FEL counterpart. (Portions of an actual FEL message, such as the TYPE 'message' feature list, are omitted to simplify the examples.)

DG_DESC* dgOpen (UCHAR* baseAddress, int interruptVector, int verbosity)

```
(execute ( (type          dgOpen)
           (name          SX_0001)
           (verbose       1)

           ; hardware base address and interrupt vector
           ; are assumed to have been determined at
           ; system initialization; so they need not be
           ; supplied by the user module
         )
)

(executed ( (type          dgOpen)
           (name          SX_0001)
           (descriptor    800140)
         )
)
```

The above example illustrates that an FEL interface to a hardware/software system can reasonably be used to insulate a module from some low level details. Once a Datacube system is configured, the base address and interrupt vector associated with a board are constant; so modules that use the system shouldn't need to know about them.

dgInit (DG_DESC* descriptor, int LUTTransformValue)

```
(execute ( (type          dgInit)
           (name          SX_0002)
           (descriptor    800140)
           (transformType unsigned)
         )
)

(executed ( (type          dgInit)
           (name          SX_0002)
         )
)
```

Notice that the DESCRIPTOR, returned as a result in the dgOpen example, is supplied as an argument in the subsequent examples. Even this level of detail could be hidden from the module, if desired.

dgConstLut (DG_DESC* descriptor, int LUTvalue)

```
(execute ( (type          dgConstLut)
           (name          SX_0003)
           (descriptor    800140)
           (value         255)
         )
)
```



```

        (executed ( (type          dgConstLut)
                    (name          SX_0003)
                    )
        )

```

As these examples illustrate, it is very easy to construct an FEL interface for a system that can be controlled via a series of parameterized functions. The function names become values of the TYPE attribute in an EXECUTE sentence and the parameters to the functions are supplied by other attribute-value pairs.

dgWtRlut (DG_DESC* descriptor, SCHAR* dataBuffer, int bufSize, int startAddress)

```

        (execute ( (type          dgWtRlut)
                   (name          SX_0004)
                   (descriptor    800140)
                   (data          (1 2 3 4 5 ...))
                   (start         0)

                   ; note: the module will count the values, put
                   ; them into an appropriate buffer and call the
                   ; C interface with the correct parameters
                   )
        )

        (executed ( (type          dgWtRlut)
                    (name          SX_0004)
                    )
        )

```

This is another example where the FEL interface is simpler than the underlying C interface supplied by the board manufacturer. In this example, a series of numeric values are written into a lookup table in the hardware.

dgRdRlut (DG_DESC* descriptor, SCHAR* dataBuffer, int bufSize, int startAddress)

```

        (execute ( (type          dgRdRlut)
                   (name          SX_0005)
                   (descriptor    800140)
                   (count         256)
                   (start         0)
                   )
        )

        (executed ( (type          dgRdRlut)
                    (name          SX_0005)
                    (data          (1 2 3 4 5 ...))
                    )
        )

```

In the final example, a series of numeric values are read from a lookup table in the hardware.

5.2. Protocol Programming

Protocol programming is the production of instructions that control the interface between two modules, apart from the internal functionality of the modules. In languages such as Pascal and C the protocol between two procedures or functions is fixed at compile time and consists of compiler generated instructions for passing parameters and returning function results. In FEL, the protocol between modules is much more sophisticated. The examples below illustrate the capabilities of protocol programming in FEL that we believe will also be needed in the complex open architecture environment of the NGC.

5.2.1. Dialog Example

FEL was designed to support complex dialogs and negotiation between modules. The protocol that governs such interactions between modules is supported in the language by requiring a feature list of TYPE MESSAGE to appear in every FEL sentence. The NAME attribute of that feature list is automatically treated as the name of the dialog of which that sentence is a part, and all FEL sentences are treated as part of some dialog or other. An example of a simple dialog is shown in Code Example 2.

5.2.2. Temporal Relations Example

Temporal relations between a request and a response can also be handled at the protocol level, independent of module-specific code. The following example illustrates how two modules might interact at the protocol level.

```
(make      ( (type      bracket)
              (name      P01234)
              (due-at    (4:30 PM 30 MIN))
              ...
            )
            ( (type      message)
              (name      HI_0532)
              (from      HI)
              (to        PL)
            )
          )

(making    ( (type      bracket)
              (name      P01234)
              (due-at    (5:30 PM 15 MIN))
              ...
            )
            ( (type      message)
              (name      HI_0532)
              (from      PL)
              (to        HI)
            )
          )
```

```

(stop      ( (type      bracket)
              (name      P01234)
              ...
            )
            ( (type      message)
              (name      HI_0532)
              (from      HI)
              (to        PL)
            )
          )

```

Code Example 9

The four FEL sentences in the above example represent a possible interaction between a human interface (HI) module and a planning module (PL). The human interface (presumably because of some user action) initially requests the planner to make a bracket, indicating that the result is needed between 4 and 5 p.m. The planning module is unable to satisfy this request on time; so it informs the human interface that the bracket is being planned for a later deadline. This is not satisfactory; so the human interface orders the planner to stop the operation.

5.2.3. Repetition Example

Modules sometimes need to request a service to be repeated some number of times. This is a sufficiently common activity that it can be handled by the generic FEL application, ideally in the module receiving the request. The following example illustrates how two modules might interact in such a situation.

```

(make      ( (type      bracket)
              (name      P01234)
              (copies    5)
              ...
            )
            ( (type      message)
              (name      HI_0532)
              (from      HI)
              (to        PL)
            )
          )

(making    ( (type      bracket)
              (name      P01234)
              (finished   1)
              ...
            )
            ( (type      message)
              (name      HI_0532)
              (from      PL)
              (to        HI)
            )
          )

```

```

...
(making      ( (type          bracket)
              (name          P01234)
              {finished      5)
              ...
            )
            ( (type          message)
              (name          HI_0532)
              (from          PL)
              (to            HI)
            )
)

(made        ( (type          bracket)
              (name          P01234)
              {copies        5)
              ...
            )
            ( (type          message)
              (name          HI_0532)
              (from          PL)
              (to            HI)
            )
)

```

Code Example 10

The control of the repetition in the above example is relegated to the receiving module, because that's where it can be handled most efficiently. If the module-specific code is designed to handle repetition, then it may be able to eliminate recomputing results that will be the same for each repetition.

On the other hand, if the module-specific code does not know how to handle repetitions, the generic FEL application level can assume the responsibility for repeatedly asking the module-specific code to perform the desired task. The generic application can provide the functionality of repetition whether or not the developer of a module makes special provision for it.

6. Graphical Interfaces to FEL and NML (and its use)

There is a concern that once we commit to an ASCII interface language that this design decision would somehow prevent an NGC system from utilizing results found in a graphic based object-oriented languages. We give a brief description in this section of how a graphical programming system could be built on top of NML and how it might be used. Furthermore, it is suggested that the choice of a simple language such as FEL would facilitate this effort.

6.1. Views of FEL and NML

Currently, FEL is a series of ASCII messages that are passed to and interpreted by various system modules. This has several advantages for both debuggers and system designers of an NGC system over binary module interfaces (e.g., subroutine calls). Debuggers can easily read and track messages that flow between modules, even without access to the internals of particular modules. (Note that third party software developers will jealously guard their source code and that it will not be possible for a system developer to track the progress of module internals.) For system designers, the ASCII representation of messages provides a simple standard view of data that bypasses many difficult low-level problems of how to represent data (e.g., representations of floating point numbers).

While a textual representation of messages has advantages, there are other tasks that have a preferred view (or representation). Various system development and debugging tools can provide the user with various sorts of "magnifying glasses" that allow various users to see the message data in the most appropriate way.

6.2. An Iconic View

Many modern programming environments are iconic and this is becoming the expected norm (witness the Apple Macintosh™ and NeXT™ machines). The idea is that the textual representation of messages would be flowing between modules and that a user module could watch these messages and display appropriate graphics from above.

With this iconic view of message data, it would be convenient to build a system wide monitor, where each module is identified with a particular graphic. As NML messages are sent back and forth between modules, this would be indicated on the lines that connect the modules (e.g., arrows could be added to represent the direction of data flow).

At a lower level (reached by double clicking on a module's graphic), each module would have a series of graphics to represent its task and its performance. The symbolic and numeric data found in the NML messages could be used to fill in the key parameters that control the overall look and settings of the graphics. For example, a message to set the furnace temperature could be sent in the usual text-form between a planning module and a control module. When this message is sent, the graphics interfaces could oversee it and then change a value in a picture of a thermometer to a new value. In many cases, you would need to have two gages to reflect the new module/device setting and the last known actual setting with a time marker indicating the time of measurement.

Eventually, this method of viewing the NGC could be used to program it as well as its eventual monitoring and diagnosis. Such graphic style of programming has become the standard in PLC's with ladder logic and more recently with Grafset for programming larger scale systems. The basic idea would be to:

- (1) draw the overall system architecture of the NGC at the module level,
 - (2) automatically have implementation modules installed,
 - (3) have available modules for an NGC dynamically offered to the system builder for generic tasks (e.g., motion controllers),
 - (4) offer design advice about any known limitations of a particular module choice,
 - (5) review the general class of NML messages that a module can deliver and receive and setup relative sources and destinations for input and output.
- (*) It would be possible for the developer of a module to use similar tools to develop a module-specific graphic interface, but it is probably not possible for the system developer to manage this task especially when source materials are being protected.

Once an NGC system was built with a graphic interface, it could then be simulated with software generated events. Finally, the NGC system would be ready to control a physical process and the corresponding graphics (used in development) can now be used to monitor process events. Diagnosis modules could also be added to safely test hypotheses for machine failures.

This vision for the next generation controller is made largely possible by the standardization of a simple interface language, such as NML. There is no doubt that it places more rigorous standards on module developers; but once this is accomplished there would be an unparalleled ability to build easy to use, robust control systems.

7. Efficiency Issues with FEL

The NGC is meant to be a controller for real-time applications and thus it must deliver task performance both relatively fast and on time. Since, NML is the message-oriented glue that binds the various modules of the NGC together, the transport and interpretation of these messages must keep up with the inherent needs of every module. However, it should be noted that a "slow" module is not necessarily the fault of NML but may indicate a poor choice of module decompositions.

7.1. Performance of FEL

In the current implementation of FEL, messages are transmitted between modules in ASCII and converted to a binary representation for use by the modules. The following chart shows typical times for messages between modules on 1 Sun 3/60, although the times between multiple machines are basically the same.

<u>Message Situation</u>	<u>Timing</u>
Typical network transport time (process to process via TCP-based sockets on 1 Sun 3/60)	0.0220 secs/msg.
Typical time in generic application (parsing and delivery to task)	0.0085 secs/msg.
Typical module to module messaging limit	32.8 messages/sec.

Note: These times are sensitive to many parameters that relate to the message sent, the machines on which the modules run and the network that they use to communicate. Therefore, these times should be used as relative timings rather than as an absolute performance measure. For example, message transport times are not tightly related to message size, but rather to the packet size of the local area network.

We believe that except for the most demanding applications that this performance is adequate. Furthermore, that by the time the NGC is a commercial possibility that it would be conservative to expect these times to be improved by an order of magnitude with no implementation changes.

7.2. Implementation Changes If Performance Must Be Increased

There are several drawbacks to making changes in the representation of a message. For examples, we consider two likely candidates for improving performance:

7.2.1. Global Binary

On a single machine it would be relatively easy to manage a global communication pool that would allow messages to be inserted in their underlying binary representation. Then, when a message is sent between two modules on a single machine it would only be necessary to pass a handle (or index) into this global data.

Comments: Managing the global data is not time-free, since solutions to: memory allocation, memory fragmentation, garbage collection and module synchronization all must

be implemented. Furthermore, this would entail slightly higher message access times and would not change the performance between machines. In addition to these issues, the engineering complexity of the overall system goes up considerably, probably making it much harder to add new modules (old time Fortran programmers know that the time maintaining global (or common data) can be excessive relative to other code maintenance time).

7.2.2. Remote Procedure Calls

A different strategy entirely would be to try and use procedure calls across module boundaries. This path would almost certainly increase performance within a machine and machine class (i.e., Suns). However, it sacrifices the possibility of expanding the basic language beyond simple subroutine calls. Furthermore, it places a strong limitation on the machines that can be considered NGC compatible. We consider these limitations too stringent to consider further this option.

7.3. Conclusions

We believe that the cost of adding implementation complexity to NML outweighs potential performance improvements. For example, it is possible that the time it takes to implement a good solution to the "global message strategy," listed above, would delay the commercialization of an NGC and in that same time it could be expected that speed improvements in available machines could rival the performance increase. Unfortunately, solutions such as accessing global data, or even RPCs, are highly machine and operating system dependent. So it may not be easy to move to the new-higher performance platforms. The software industry has encountered this very issue countless times.

8. Enhancements to the Generic FEL Architecture

Several enhancements to the generic FEL environment are proposed to enable FEL to satisfy identifiable requirements for NML. In this section we look at those requirements, the problems of the current implementation of FEL with respect to those requirements, and propose feasible solutions.

8.1. Real-time Support

By "real-time" we mean "fast enough for the purpose at hand." So a computation that requires several hours to complete will be said to occur in real-time if, on that particular occasion, waiting several hours for the result is fast enough for our needs on that occasion. For a human user of a time-sharing computer, quarter second response is usually considered real-time, although high-powered personal computers have been raising peoples' expectations.

In order to be able to provide real-time response, a computer system must be able to respond on demand within a guaranteed time interval. To interact with such a system, a language must be able to designate when a response is required, possibly within a time interval.

8.1.1. Adding Support for Time

There is no built-in support for real-time processing in the current version of FEL, except for representing values with temporal units, such as seconds, revolutions per minute, etc. Support for real-time processing can be added to FEL by:

- (1) adding some new attributes to the language; and
- (2) enhancing the generic FEL processing environment to handle the module independent aspects of time.

8.1.2. Attributes for Time

Two new attributes can be added to FEL that will allow for flexible specification of when something, such as the completion of a process, is to occur. These attributes are DUE-BY and DUE-AT, as illustrated in Figure 3, page 36. The bracketed portion of each time line shows the range of times satisfying the indicated request. Saying that something is due by a certain time t , plus or minus ϵ , means that the request should be satisfied no later than $t+\epsilon$, but it is okay for it to be satisfied immediately. Saying that something is due at a certain time t , plus or minus ϵ , means that the request should be satisfied between $t-\epsilon$ and $t+\epsilon$; earlier satisfaction is not acceptable.

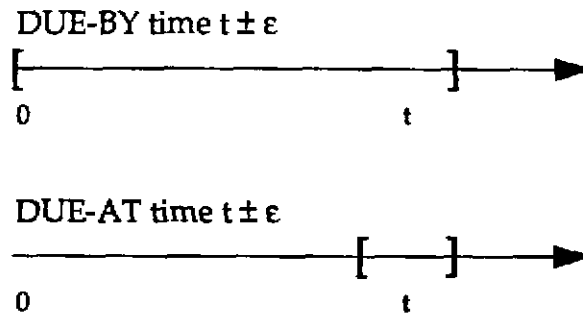


Figure 3. Two Ways to Schedule an Event in Time

In FEL, each of these attributes would have as its value a pair of real numbers, $(t \pm \epsilon)$, where the first number t would specify the desired time by which or at which an event should occur. The second number ϵ in the pair would specify the maximum allowed deviation of the time of the event from t .

8.1.3. Generic Processing of Time

The addition of temporal attributes to FEL would enable real-time modules to be usefully incorporated within systems of modules communicating with each other in FEL. But, at the very least, each real-time module would have to include code to handle aspects of temporal processing such as keeping track of various requests and when they need to be responded to, informing other modules when a request cannot be handled within the requested time, storing computed responses that have to be delivered later and delivering them on time, and notifying other modules of the minimal or expected time needed to satisfy a request. Some, or all, of these chores could be handled, in whole or part, by an extension of the generic FEL module environment (see the subsection "Generic FEL Architecture" under "Current Status of FEL").

8.2. Dialogs

8.2.1. More Support for Dialog Management Needed

Further work needs to be done on the protocol level in the C++ version of the generic FEL environment in order to provide better support for dialogs.² Currently, it is up to each module implementer to keep track of the relationship between messages with the same dialog name. It is possible to largely automate this.

8.2.2. Possible Improvements

Within the context of our generic application for modules written in C++, the following improvements could be made in the way dialogs are managed:

- (1) Dialogs can be represented as separate tasks using the C++ task system and thereby take on an added level of independence within a module.

2. The level of support being suggested here largely exists already for the Lisp version of the generic application.

This representation would provide the implementer a natural place to maintain dialog specific information.

- (2) The creation of dialog tasks can be automated for dialogs initiated by another module and mostly automated for dialogs initiated locally.
- (3) In the process of automating dialog creation, the generic application can be enhanced to automatically forward incoming messages to the appropriate dialog.

It is not necessary that the C++ task system be used. The advantage is that it provides a straightforward mechanism for representing multiple dialogs in such a way that a dialog may be suspended and later resumed, e.g., upon receipt of a subsequent message or after a specified delay.

8.3. Language Extensibility

8.3.1. The Problem of "Compiled In" Names

The current version of FEL has a centrally maintained parser with "compiled in" lists of verbs and attributes. This has several potential advantages, including:

- (1) the language is guaranteed to be syntactically the same in all modules using the common parser;
- (2) the growth of the language is more likely to be conceptually compatible with previous design ideas implicit in the choice of verbs and attributes, if one person is in charge of changes;
- (3) the chance of error is reduced and the ability to fix errors is enhanced.

Unfortunately, having "compiled in" lists of verbs and attributes also has some serious disadvantages, including:

- (1) it hinders the distributed development of new modules that require new verbs and attributes;
- (2) any effort to experiment with new sentences impacts everyone, when it only need impact the modules that will use the new keywords;
- (3) someone has to deal with all changes to the language.

8.3.2. Dynamic Verbs and Attributes

We propose to simplify the addition of new vocabulary to FEL by allowing module developers to update FEL with new dynamic verbs and attributes that can be added to the parser's standard list at run time. Since all modules will share a standard "compiled in" list of verbs and attributes, a set of standard services, e.g., automated error messages, can be provided and guaranteed. Since all modules will have the ability to augment the

standard list, development efforts will not be hampered by centralized control.

We will implement dynamic verbs and attributes using files of specially formatted data that will be automatically loaded at run time.

8.4. Error Handling

8.4.1. How Errors Can Arise

There are many ways in which errors can happen in a system of modules, but errors will always fall into one of two groups:

- (1) errors that result from requests that are in some way invalid relative to the receiving module's documented (intended) capabilities; and
- (2) errors that result from the failure of module-specific code to perform as documented.

Errors of type (1), i.e., invalid requests, may be either syntactic or semantic. FEL sentences produced by the functions provided with the generic FEL application will never be syntactically incorrect, but they may be semantically incorrect in four ways:

- (1) an attribute may be assigned a value of the wrong type for the request being made (attributes do not have fixed types);
- (2) an attribute may be assigned an incorrect value of the right type for the response being supplied;
- (3) a list of attribute-value pairs may be incomplete for the request being made; and
- (4) the verb may not be meaningful to the receiving module (but will always be one of the legal verbs of the language).

Errors of type (2), i.e., module-specific errors, may result in no response being supplied to a requesting module or they may result in an incorrect response. The latter may be semantically incorrect in any of the three ways listed above.

8.4.2. Approaches to Error Handling

It is desirable for there to be a uniform mechanism for handling errors that arise within the context of the generic FEL application.

Some types of errors can be handled automatically by possible extensions to the generic application, forestalling any possibility of catastrophic failure due to a module's receiving and mishandling an erroneous message. For example, a module could be required to provide the generic application with a specification of the FEL sentences that it is designed to handle (in a non-trivial way). The generic application could then use the specification to filter messages addressed to the module, passing those that match the specification and automatically returning error messages for those that fail to match.

In the case of errors that can only be recognized by the module-specific code of a module receiving a request, we can provide standard procedures for responding to errors that will simplify the task of implementing error-handling within modules.

8.4.3. Automating Error Handling

8.4.3.1. An Easy Improvement

A simple way to (partially) protect a module from having to deal with unwanted messages is to have the generic application filter out messages that have a verb that has no special meaning to the module. This could be implemented with minimal overhead within the FEL parser itself.

8.4.3.2. Moderately Complex Improvements

Modules could provide the generic application with more detailed information about the expected contents of incoming messages. For each verb and type of feature list, a list of "required" additional attributes could be specified and the generic application could reject messages that contained feature lists with missing required attributes. This level of error checking would not involve any changes in FEL, but it would incur considerably more overhead than just checking that the verb is one that the module wants to deal with. Also, detecting missing attributes is something that happens "naturally" as a module collects the values it needs to perform the requested function.

Checking that attributes are assigned appropriate types of values is not possible without a change to FEL to allow "typing" of attributes.

8.4.3.3. Handling Catastrophic Failures

Modules based on the generic application can be made more robust against certain types of catastrophic errors. For example, within the Unix environment, some types of errors, such as division by zero, cause signals to be generated. These signals can be caught, saving the processing from terminating, and an appropriate action taken. Worrying about errors such as division by zero errors is not a fanciful exercise. The NGC will need to include sophisticated geometrical modeling capabilities; it is notoriously difficult to implement reliable geometric algorithms using finite precision arithmetic and many such programs can be rather easily be made to crash on valid input.³

The generic application already recognizes out-of-memory conditions, but does not do anything about it except for shutting down the process cleanly. The shutdown is reasonable, but it would be better to first inform other modules that it is taking place so they will know that the failed module is no longer available.

Very little can be done to automate handling module-specific errors that result in no response from a module, aside from implementing timeouts in the requesting modules. Asking a module to say why it has not responded to a request is not likely to be effective if the module was not designed to catch and report an error in the first place.

3. For a technical discussion of this problem and a proposal for dealing with it, see [2].

The generic application cannot do anything about errors that result in the wrong value (of the appropriate type) being associated with an attribute, though the results are likely to be catastrophic and unpredictable.

9. Conclusion

We have shown that in principle all of the basic functions of NML can be accomplished with a simple message-oriented language such as FEL. However, in the process of this exercise we learned several important lessons.

- (1) **Module Granularity** - The required computational power of the language directly trades off with the sophistication of the modules. This has several consequences: the control structure of a module's task must be represented within the module, hyper-real time applications should probably be left as a single module.
- (2) **Representational Adequacy** - We have considered various manufacturing applications of how FEL could be used to communicate. For example, motion control, hardware board configurations (e.g., vision boards) and part representation. With the right module decomposition, all of these applications can be properly serviced with an FEL-like language.
- (3) **Performance** - While it is possible develop schemes for enhancing the performance of a message-oriented language, there is some serious question as to whether these complex schemes are worth it. The advantage of a language such as FEL is simplicity and if this is lost in the name of performance then the ease of integration and even the advantages of an open architecture begin to be compromised. Furthermore, a simple language can be processed very quickly and this will improve further as new platforms for the NGC evolve.
- (4) **Representing Time** - The NGC has an unambiguous need to obey time constraints. Therefore, to properly service this need NML must have a representation of time that is globally accepted by the modules.
- (5) **Dialogs** - The requirements of NML do not seem to recognize the need for "back-and-forth" communications between modules. A language that only supports "one-message-at-time" is essentially equivalent to open loop control. But we know from many years of research in Artificial Intelligence that virtually every kind of decision requires some level of feedback. This could be as simple as "Do a task - OK" to "Do a task - I can't and here's why" to "Can you do a task this way - no, but how about this - OK but make sure you do that - OK". Furthermore, these discussions may be held in conjunction with other multi-party dialogs.
- (6) **Standardized NML Front Ends** - Even in this simplest variations of NML, there is a complexity that must be managed properly and equivalently by each module. Rather than allowing a module developer to pick and choose what NML features they will implement there must be standard package that can be utilized by ALL modules to support ALL of the NML functions.

A successful NGC program is heavily dependent on the choice and design of NML. The acid test will be how difficult it is for third party to developers to use NML to integrate their products. If it is very easy to use NML and to be NGC compatible then the NGC program will have the best chances for success.

Bibliography

- [1] Bourne, David Alan, Jeff Baird, Paul Erion, and Duane T. Williams, *The Operational Feature Exchange Language*, Carnegie Mellon Robotics Institute Technical Report, CMU-RI-TR-90-06, March 1990. This is a compilation of several reports on FEL, including the original design document and the FEL interface descriptions for key IMW modules.
- [2] Milenkovic, Victor Joseph, *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*, Carnegie Mellon University, Dissertation in Computer Science, CMU-CS-88-168, July 1988.

Appendix A: Commentary on the NGC Requirements Definition Document

This appendix is a commentary on the requirements in the *Next Generation Workstation/Machine Controller (NGC) Requirements Definition Document (RDD)* that pertain specifically to NML. In each case, the requirements and associated RDD comments are quoted, followed by an assessment of how well FEL meets these requirements.

1. RDDR-56

REQUIREMENT: The SOSAS shall make provision for entering and displaying NML sequences in human understandable form using graphic and textual presentations appropriate for the specific user.

COMMENTS: Because an NML statement may apply to a range of machinery and processes, it is important to provide the human user of NML with Context-dependent interpretations of NML statements and sequences.

FEL is currently presentable in a textual form suitable for use by module developers. Clearly, the information conveyed in FEL sentences could be presented in other graphic and textual formats.

2. RDDR-87

REQUIREMENT: The SOSAS shall make provision for a language in which all communications with, and within, an NGC can be expressed. This language shall be named the Neutral Manufacturing Language (NML).

How well any language satisfies this requirement will depend upon what communications need to be expressed and that will depend on the implementation of the modules. Although the syntax of FEL is simple, its expressive power is considerable.

3. RDDR-88⁴

REQUIREMENT: The SOSAS shall make provision for expressing in NML those elements which an end-user will use in communicating with an NGC.

COMMENTS: These elements will include: descriptions of processes and their successive levels of decomposition, product descriptions using features and attributes, and equipment descriptions including the capability, capacity, and topology of the workstation. Other elements will be found in Figure 3-1 (of the RDD).

Section 5 of this report includes examples that demonstrate the flexibility of FEL for communicating with various kinds of modules, including process and product descriptions.

4. This requirement is beyond the scope of the Air Force NGC Program.

4. RDDR-89

REQUIREMENT: The SOSAS shall make provision for NML statements which have equivalent meanings across diverse NGC configurations, types of manufacturing equipment, and equipment-specific processes.

COMMENTS: Example equipment includes machine tools, robots, and dimension-measuring equipment. The NML statement "clamp workpiece" is meaningful to each of these types of equipment.

FEL is capable of accommodating this requirement. In fact, a design goal of FEL was to keep the number of verbs and attributes relatively small, and to make them sufficiently general so that they could be used to express conceptually similar operations in multiple contexts.

5. RDDR-90

REQUIREMENT: The SOSAS shall make provision for expressing in NML process descriptions which can be implemented on a wide range of machinery using a wide range of processing techniques.

FEL is compatible with this requirement. It is currently implemented on two very differing architectures: a Lisp machine and a Unix machine in C++.

6. RDDR-91

REQUIREMENT: The SOSAS shall make provision for expressing in NML statements pertaining to every level of the SOSAS architecture.

COMMENTS: The SOSAS should make provision for expressing in NML:

- Upper levels of manufacturing processes, such as machining, assembly, and inspection
- Intermediate levels of manufacturing processes, such as turning, milling, and grinding
- Lower level of generic processes common to manufacturing processes, such as continuous motion control, discrete logic control, monitoring, diagnostics, model definition, and material handling
- Low level activities such as move, latch and activate.

These "higher-level" descriptions are the basis for translation into more machinery and process-specific descriptions, which are also expressed in NML.

These descriptions are applicable not only to a range of machinery and processes which might be used presently in an installation, but they are

applicable also to upgraded machinery and processes. Thus, they greatly reduce the cost of upgrading.

FEL is not level-specific in any way; so there is nothing intrinsic to FEL that would prevent its satisfying this requirement.

