Tool Extension in an ALOE Editor
(M.S. Dissertation)

Takahisa Ishizuka
Columbia University
Department of Computer Science
New York, NY 10027

thesis committee: Profs. Gail E. Kaiser and Peter Allen

September 1988

CUCS-368-88

Abstract

This technical report consists of an M.S. thesis that presents the development of a technology for integrating language-based editors with existing tools, whereby the editor and tools operate as cooperating processes communicating *via* message passing. The thesis describes the validation of this technology with the design and implementation of A Manufacturing Programming Environment (AMPE), which integrates an ALOE language-based editor for AML/X with the AML/X interpreter. The dissertation includes the user manual for AMPE as an appendix.

AMPE was funded by IBM Contract #703013, Research in Robotics and Manufacturing.

TOOL EXTENSION IN AN ALOE EDITOR

MS Thesis

thesis committee: Profs. Gail Kaiser and Peter Allen

Takahisa Ishizuka
Columbia University
22 September 1988

Abstract

Many approaches to tool integration can be seen as variations on two models, the "sequential" and "concurrent". "Sequential" models are characterized by a high degree of modularity. In this model, tools execute in sequence and work on separate data structures. "Concurrent" models are characterized by the ability to interleave the execution of various tools. Here, tools work on a common data structure. This thesis describes AMPE (A Manufacturing Programming Environment) which consists of two tools integrated in a way that has advantages from both models. The execution of both tools can be interleaved, but they use separate data structures and maintain a degree of modularity. AMPE consists of an ALOE structure editor, which has been extended to work with an interpreter for the programming language, AML/X. The two tools run as separate processes and communicate through message passing.

Table of Contents

1. Introduction

ALOE (A Language Oriented Editor) editors [5, 7] provide useful interactive support in the development of programs. As structure editors, they manipulate program trees and thereby eliminate some of the more tedious aspects of programming, such as matching *begin* and *end* statements and placing semi-colons correctly, while at the same time enforcing the syntactic correctness of programs. They also provide the means to do static semantic checking, which can be done automatically so that the programmer does not have to worry about it, and done incrementally so that the programmer does not notice it occurring except when there are errors. Dynamic semantic checking can be provided by internal routines, often in the form of daemons, that can be used to implement an interpreter and also debugging facilities. Such a programming environment eliminates the need for a parser and provides for other programming tools such as an interpreter and debugger in an integrated fashion.

Although the integration makes the system convenient by providing easy access to a set of programming tools, there can be problems if the system does not already include those tools that a programmer prefers to use. For example, if a programmer has a favorite debugger outside of ALOE, he is not able to use it. Also, in order to interpret code, an interpreter has to be written from scratch even when an interpreter already exists for the language. Therefore, an ALOE editor should not be thought of as a completely self-contained unit, but as one that can also potentially be integrated with a variety of existing tools.

This thesis explores how outside services can be used in the context of a structure editor. In particular, the work for this thesis explores how the ALOE editor, AMPE (A Manufacturing Programming Environment), was designed and generated for the programming language AML/X [10]. Although it is possible to generate ALOE editors that execute interpreters internally through the use of implementor-defined daemons, it was decided that the ALOE editor would communicate with the AML/X interpreter running as a separate process to do evaluation. The reason is to allow the editor to work with an interpreter that could be running on a separate machine under a possibly different operating system. This forces a clean interface between the editor and interpreter, and at the same time saves us from having to reimplement the interpreter inside the ALOE editor.

The two components of AMPE, the editor and interpreter, can be viewed as working in a *dual client-server* relationship. That is, the components may work together in client-server relationships where each can be client or server at various times. For example, two such relationships are:

1. **Interpreter as an Evaluation Server** - The editor is the client and makes a request to the interpreter to evaluate some code.
2. **Editor as I/O Server** - During the evaluation of code, the interpreter needs to read a file that is remote to itself and local to the editor. The editor then must pass the necessary data to the interpreter so that it can complete its evaluation.

When thought of in this way, the design and implementation of AMPE can be thought of as defining:

1. **Protocols and Interfaces Between Clients and Servers** - Protocols for making requests and sending data must be established. In order to use these protocols, it may be necessary to transform data, either before sending or receiving it.
2. **Services to be Offered** - The routines that provide the services that server routines offer to client routines.

The following section provides some background to the work done on AMPE, explaining what ALOE

editors are and how they are generated by the *Gandalf* system. Language features of AML/X that are relevant to debugging are also discussed. Section three discusses the issues encountered in designing AMPE. The major issue is how to have a structure editor communicate with the interpreter in some effective way. Section four describes the actual implementation of AMPE. Section five provides examples of how AMPE can be used. Section six discusses related work and compares AMPE with "sequential" and "concurrent" models of tool integration. The last section states conclusions drawn from this work.

2. Background Work

Because of the separation of ALOE editor and AML/X interpreter in AMPE, all run-time support must be provided by the interpreter. In the case of AML/X, there are already good debugging features and facilities for creating new ones. The existing debugging features together with the AML/X interpreter already provide a way for doing dynamic semantic checking and provide run-time support. However, static semantic checking, and the enforcement of syntactic correctness is not supported, and an ALOE editor is well suited to remedy these deficiencies. Furthermore, the use of the existing interpreter saves the ALOE implementor from having to reproduce the functionality of the AML/X interpreter (which is rather large) in the ALOE and at the same time preserves a sense of familiarity for programmers who previously used the interpreter directly. Therefore, the functionality of the ALOE editor, as a front-end, and AML/X interpreter complement each other well.

2.1. Gandalf

Gandalf is a system for generating interactive programming environments. It consists of several components that allow the implementor of the environments to design and fine-tune the environments. *Gandalf* supports *programming-in-the-large* and *programming-in-the-small*. *Programming-in-the-small* is supported through *ALOE* editors.

2.1.1. ALOE Editors

The problem of constructing a pure syntax directed editor is already well understood and ALOEGEN provides a very straight forward method of doing this, given an abstract syntax and unparsing schemes. The utility of ALOE editors in providing syntactic and semantic checking has already been demonstrated by existing systems [5]. Static semantic checking is also provided for through the use of daemons, which are written in the action routine language, ARL [12]. ARL daemons can be declared to be associated with a particular grammar production and are activated when a node of the declared type is edited. When activated, the daemon does the semantic processing appropriate for that node.

ALOE editors present the programmer with a uniform method of entering commands. Commands to edit the tree, such as those to create and delete nodes, and those for cursor movement are executed with *emacs-style* keystrokes. In addition to the edit commands, ALOE editors also provide for implementor-defined *extended* commands, which are executed with similar keystrokes, but are chiefly used for interpretation and debugging.

The following is an example of how the ALOE editor part of AMPE works for AML/X:

An AML/X program consists of a list of statements and declarations, which will be constructed by filling in templates. Upon entering the editor, the programmer sees a template indicating a program component

that needs to be filled in. Program templates are called *meta nodes* and are represented as the name of the components prefixed with "$". The underlining of "$stat_decl" indicates the current cursor position.

```
$stat_decl


Window:root Node:META          Class:stat_decl
>
```

In AMPE, a *stat_decl* may be a *statement* or *declaration*. Therefore, a *declaration* command has been defined. This command corresponds to the *declaration* production in the description of AML/X, which was used to generate the editor. In order to execute the command, the cursor is moved to "$stat_decl", and *declaration* is typed at the ">" prompt, at the bottom of the display. The screen is then updated to show:

```
$label $declarator

$stat_decl


Window:root Node:META          Class:label
>
```

"$stat_decl" has been replaced with "$label $declarator" because a declaration in the language description of AML/X indicates that a *declaration* is a *label* followed by a *declarator*.

After the templates for the declaration are filled, and the programmer attempts to leave this subtree, the implementor defined semantic checking daemon (if one exists) associated with the *declaration* subtree is activated automatically. The programmer is then notified of any static semantic errors, an example of which might be the reuse of an already declared variable name, since in AML/X, variables may never change type. Currently semantic checking routines are not implemented in AMPE.[1]

2.1.2. Components of Gandalf

The ALOE component was developed within the *Gandalf* environment generation system, which consists of the following five system components [9]:

1. **SMILE** - which supports *programming-in-the-large* and *programming-in-the-many* by providing a multi-implementor, multi-module programming environment.
2. **ALOEGEN** - which provides a syntax development environment. It assists the ALOE implementor in defining abstract and concrete syntax. ALOEGEN is itself an ALOE editor that supports *programming-in-the-small*.
3. **ARL** - which provides a semantic description formalism. It is an imperative language that works on ALOE trees, and is used for semantic analysis and other operations requiring access to ALOE trees, and access to the various data fields of these trees.
4. **DBGEN** - which provides the means of putting the different parts of the editor together and also provides the means of fine tuning editor functions that are unrelated to any ALOE program trees.
5. **the ALOE kernel** - which provides primitives for ALOE tree manipulation and terminal I/O routines.

---

[1] Semantic checking routines have not been implemented because they are not related to the issue of tool integration.

As *SMILE* can be used just as a C programming environment, it is possible to develop C code that works with ARL routines. It is also possible to link the editor code with C libraries.

2.2. AML/X

AML/X is an *expression-oriented* language and an experimental version of the robotics programming language, AML. While AML/X's origins are in robotics, AML/X is intended to be a general purpose programming language.

Run-time support and debugging facilities are provided by the AML/X interpreter. AML/X already provides an interactive debugging environment. AML/X also comes with a set of standard debugging commands [10] which can be augmented with new, user-defined commands.

2.2.1. Debugging Facilities

Existing AML/X debugging facilities are used. AML/X already provides debugging commands that allow the programmer to:

1. **Stop Execution** - there are commands for setting breakpoints.
2. **Display and Assign** - once execution is stopped, any variable that is active or defined may be displayed or have its value changed.
3. **Trace Program Execution** - the execution of various statements may be traced during execution.
4. **Altering the Programming Environment** - the interpreter's environment can be altered without restarting the program.
5. **Create Debugging Scripts** - debugging scripts can be stored in a separate file that can be loaded when a particular application needs to be debugged.

These built-in debugging commands are implemented using a built-in primitive command that raises exceptions. This primitive command is available to the programmer to use in implementing his own customized debugging commands.

2.2.2. Environment Variables

A programmer may want to examine how different pieces of AML/X code behave in a particular environment, say in some block with local variable declarations. We have to be able to handle such cases in order for the system to be useful in practical applications, where it may not be possible to restart a program due to its size or its dependence on certain sensory data (as in robotics applications), which are difficult to replay. We will also need to be able to control and modify the context of execution when debugging. This is already possible using AML/X debugging routines, AML/X *environment* variables.

AML/X provides an *environment* data type and functions to manipulate environment variables. Among the functions that manipulate these block execution environments are functions that return the current or previous environments, and a function that returns the names of the variables known in a specified environment. There is also a built-in *intern* command that allows an identifier to be evaluated within a given *environment*. These variables and functions extend the power of the preceding debugging commands.

3. Design Issues

In combining the ALOE component of this environment with the AML/X interpreter, we need to address three issues. These are:

1. **Generating AMPE** - We need to decide how to generate an editor that can use the existing code for the AML/X interpreter.
2. **Evaluation of AML/X Code** - We need to define the functions that allow an ALOE tree to be evaluated by the interpreter. At the same time, we also want to be able to load existing AML/X programs that already exist in text form.
3. **Operating System Interface** - This includes problems relating to IO that occur when linking the editor and interpreter. This is not so much an issue for the ALOE component, because most of the needed commands are generated with the editor, but file I/O required by the interpreter must be provided for. Clearly the editor needs to replace standard I/O in the interpreter. Since the interpreter may run on another machine, file I/O may need to be channeled through the editor.

These three issues will determine how the editor and interpreter will communicate and will help define the services that each will offer the other in AMPE.

3.1. Generating AMPE

When generating AMPE, a fundamental problem that has to be resolved is that the two components, the ALOE editor and AML/X interpreter, use different data structures. Therefore a way of integrating the two components must be found. There are two ways of doing this:

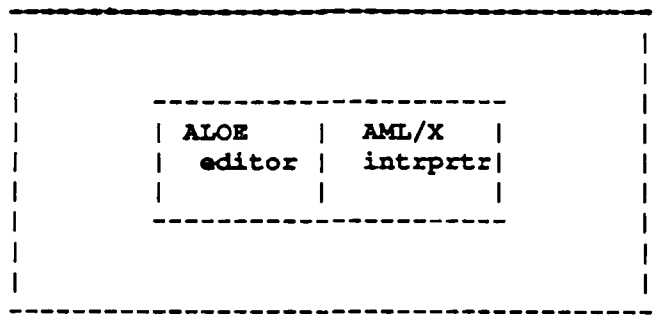
1. **Embed the Interpreter** - Generate the editor using the interpreter's code inside the ALOE editor and call the appropriate interpreter routines directly.
2. **Keep the Interpreter Separate** - Generate the editor separately from the interpreter and run the interpreter as a separate process and communicate with it through message passing.

In both cases we need to translate ALOE program trees into parse trees that can be used by the interpreter. The method used for this translation would be the same in both cases. Evaluation occurs in AMPE in the same way similar to the way the original AML/X interpreter would have evaluated. This is because regardless of how we integrate the editor and interpreter, in the end, the interpreter evaluates the same parse tree.

3.1.1. An Embedded Interpreter

We could have embedded the interpreter in ALOE. In order to allow ALOE to link to the interpreter's code, it is possible to have turned the interpreter into a C archive. One obvious problem with this approach in the general case is the possibility that there would be identifier conflicts between the editor and interpreter, particularly for functions that have a very basic functionality, such as *hash* and *length*. The *Gandalf* system and AML/X interpreter had only two such identifier conflicts, which were the functions just mentioned. In this particular case, these conflicts would not have been a problem to fix due to the way the interpreter was written.

AMPE with Embedded Interpreter

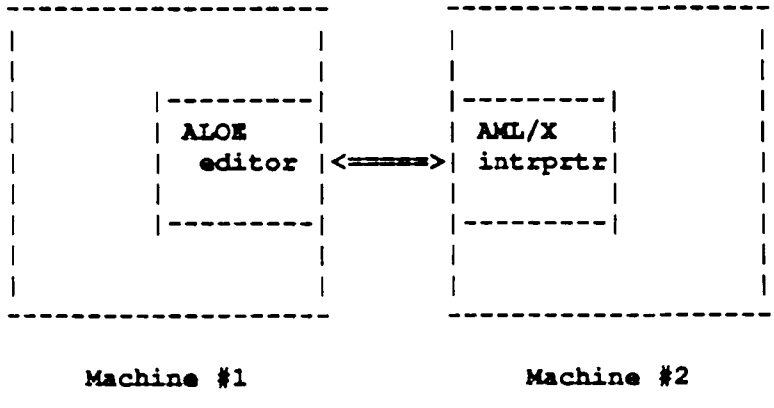


Once the editor and interpreter have been linked together, we could have rewritten the interpreter's *parse* function to call ARL routines which in turn call the appropriate interpreter routines that do the parse. This achieves the goal of not having to rewrite the interpreter from scratch, but tends to decrease the modularity of having separate editor and interpreter components.

3.1.2. Interpreter as a Separate Process

The editor can be linked to the interpreter via sockets, as is the case in AMPE. The simplest way to connect the editor and interpreter would be to use the ALOE unparse schemes to transform the program tree into text form. The text can then be written to a socket. The interpreter, which already expects code in text form, then reads from the socket and evaluates. The problem with this approach is that it is inefficient to unparse a tree in ALOE just to have it reparsed by the interpreter. Some mechanism for converting an ALOE tree into a tree usable by the AML/X interpreter is needed. A more efficient way to do it would be the following. When evaluation of an ALOE tree has been requested, the editor makes a preorder traversal of its program tree and sends messages to the interpreter. The interpreter meanwhile uses the information received from the editor to build its own parse tree for evaluation.

AMPE Running in two Processes



There are advantages in this approach over the previous one. First, this approach provides greater utility. The interpreter can run as a separate machine under a different operating system, provided sockets are supported.

There are also advantages in ease of implementation and maintenance. The problem of conflicting

identifiers encountered in the embedded approach is avoided. Also having the interpreter run in a separate process forces a cleaner interface between the editor and interpreter which makes it easier to modify the interpreter without regenerating the editor part. This clean interface has another advantage. In the embedded approach, all phases of interpretation must be explicitly provided for. For example, if we want to evaluate the parse tree, we must explicitly call the evaluation routine. If we want to display the interpreter's *prompt*, we must call the appropriate routine explicitly. In a sense, we would be reimplementing the interpreter in the editor, although at a higher level of abstraction. With the interpreter running in a separate process, all these things are done automatically for us, with the changes required to the interpreter's source code being localized.

3.2. Evaluation of AML/X Code

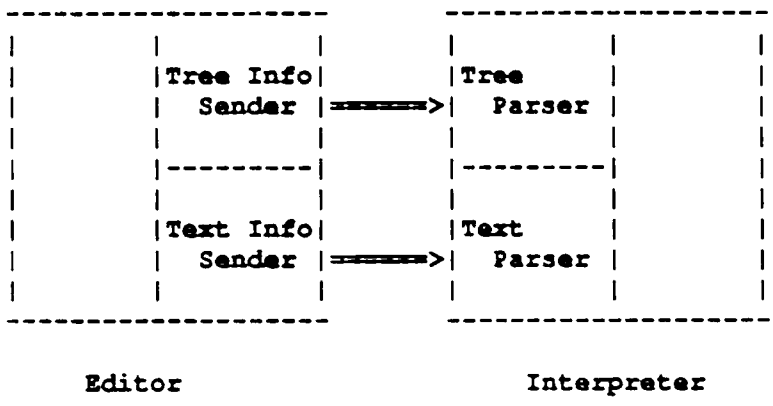
Having decided that the editor and interpreter will run in separate processes, we need to decide how the editor can use the interpreter. Since the editor is the interface to the interpreter, the interpreter will initially be waiting as a server. When the user of AMPE wishes to evaluate some code, an evaluation request is sent to the interpreter.

AMPE must be able to evaluate two kinds of AML/X code:

1. **Tree Code** - The interpreter will have to be able to evaluate code that has been developed in AMPE and therefore exists in the form of an ALOE tree data structure. ALOE will need to send information about program trees to the interpreter.
2. **Text Code** - The interpreter will have to be able to evaluate code that has been developed outside of AMPE in a text editor and therefore exists as a file of characters. Existing AML/X program libraries will probably already exist as text files. ALOE will need to send text files to the interpreter.

AMPE must clearly be able to do both kinds of evaluation. It must be able to evaluate ALOE trees, since code developed in AMPE must be testable in AMPE. It must be able to evaluate AML/X code in text form since a programmer should be allowed to make use of and add to existing AML/X libraries.

Needed Tree and Text Evaluation Routines



The *evaluation* command can be used to allow the programmer to enter an arbitrary AML/X expression to be sent to the interpreter. This allows the programmer to take full advantage of existing and future AML/X programming tools that are run by the interpreter.

3.2.1. Evaluation of Trees

When evaluating trees, an issue that must be addressed is how to have the AML/X interpreter evaluate the program tree used by the ALOE editor. The basic problem is that the editor and interpreter use different data structures. Therefore some translation of data structures will be necessary. This translation is accomplished by having the editor traverse its program tree while sending messages to the interpreter. The interpreter uses the information from the editor to construct its own parse tree.

In order to allow a programmer to experiment with different alternative subtrees, we will also have to evaluate subtrees. This is not a problem because AML/X, like LISP, is an *expression-oriented* language. The extended command, *evaluate*, mentioned earlier, will execute the AML/X code represented by a subtree of the main program tree, as well as the main program tree itself.

There are two kinds of evaluation commands that should be offered to the programmer:

1. **Evaluation Command** - The first type is for the evaluation of ALOE program trees, using an *extended* command. This command causes the expressions indicated by the user's cursor to be evaluated by the interpreter. It allows an arbitrary AML/X tree to be entered and evaluated, providing unrestricted access to the interpreter.

2. **AML/X Language Commands** - The second type involves implementations of a few of the standard AML/X commands as ALOE *extended* commands. In this case, we do not have an actual program tree to evaluate. We evaluate an AML/X command without traversing an actual existing tree. Instead, the editor simply send messages to the interpreter as if it were traversing a tree. The interpreter does not know the difference. This approach is more convenient that having the programmer create a subtree using the ALOE editor commands, evaluate it, then when it is no longer needed, delete it. Since we know that AML/X commands are *application_expressions* (similar to *functions* in other languages), we can just pretend to traverse an *application_expression* tree and thereby execute the AML/X language command.

Some of the AML/X language commands that we would like to be able to execute through ALOE *extended* commands would be:

1. **debugging commands** - AML/X debugging routines.

2. **load** - this command loads a given file into the interpreter.

AML/X provides a small set of standard debugging commands, which we will rely on to provide debugging support. It would be difficult to implement debugging commands that are not run by the interpreter, since it runs on a separate process and will not share a common data representation with the rest of the system. Each of these may be implemented as an extended command so that it does not have to be typed each time the programmer wants it executed. But in addition to these commands, AML/X also provides for user-defined debugging commands, so we could allow for the possibility that users will want to use their own debugging functions. One approach to doing this would be to supply AMPE with a set of predefined *system* variables. *Extended* commands will be implemented that execute each one. User-defined routines could be executed by these *extended* commands by assigning them to these *system* variables. The effect is to implement what are essentially *function keys* in the editor.

When possible, arguments to AML/X commands should be passed automatically. An example of this would be when setting breakpoints. Suppose the programmer wants to set a breakpoint at the current cursor position. The interpreter needs a line number in order to set the breakpoint. Such line numbers can be automatically provided through routines that automatically traverse the program tree. This is one

of the advantages of working with program trees rather than text. But when the original AML/X parser parses text, it gathers information about the text, such as information about line numbers. This kind of information must be accounted for when information about program trees is sent from the editor to interpreter.

Some commands such as *load* raise special problems. *Load* requires AML/X text files to be loaded to the interpreter. This raises some issues about how interpreter I/O should be handled. One of the problems is that the interpreter can be running on a machine different from where the editor is running and where the needed file resides. This problem is discussed in section 3.3.

3.2.2. Evaluation of Text

In this section we discuss the issues of using text code in AMPE. ALOE editors work on abstract syntax trees but there is a need for AMPE to be able to load and generate AML/X in text form. Generating a text file of a program tree is already possible using the unparsing schemes in the ALOE editor. The *unparse* command that does this is generated automatically by the editor generator. Therefore, it is a relatively simple matter to be able to add to existing subroutine libraries.

However, there will be problems when using AMPE with text files. This can happen we want to load text files into:

1. **Editor** - The editor works only on abstract syntax trees. It is unable to convert text files into the necessary data structures.
2. **Interpreter** - The interpreter needs to be able to read text that is remote to itself and local to the editor. It must also be able to parse and evaluate text code, as when executing the AML/X *load* command.

The problem of being able to execute the AML/X *load* command shows the need to be able to process AML/X code in text form. Given an evaluation mechanism (it already exists in the original AML/X parser), we can send lines of text from the editor to interpreter for evaluation.

Currently, ALOE editors read only code generated by the ALOE editor, restricting its general use. The problem is that ALOE editors manipulate tree structures and do not parse the plain text. Therefore, the ALOE editor could not be used on code in existing subroutine libraries which were generated by text editors. The problem would be in implementing the following desirable command:

1. **edit** - this command transforms a text file into a program tree that is then loaded into the editor for editing.

When modifying code that exists already in textual form, it must first be transformed into a tree structure acceptable to the ALOE editor. Since the ALOE editor stores only information about the abstract syntax of AML/X, it is unable to parse AML/X text. Even if parsing were possible, there is the problem of how to handle comments in the original text. Comments would have to be stored and unparsed again after editing. It seems that distortion of program text during conversions between tree and text form is inevitable.

When executing the AML/X *load* command, the interpreter will need to read, parse and evaluate AML/X text if the file to be loaded is a text file. If the file is remote to the interpreter, the editor may have to send it. Therefore, there will be a problem in executing the following AML/X language command:

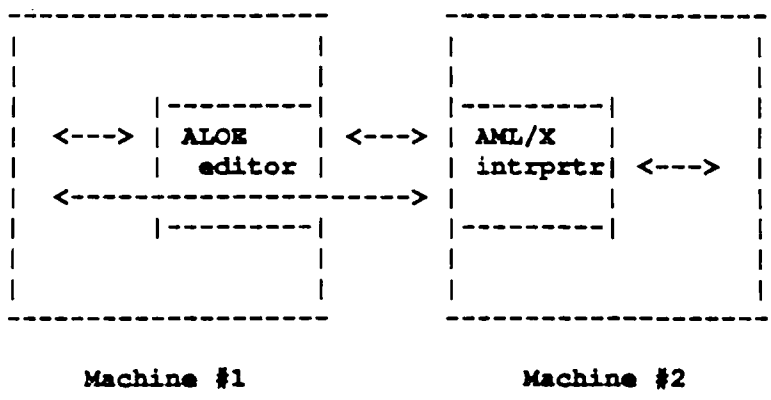
1. **load** - this command loads an AML/X file into the interpreter.

If an ALOE tree file needs to be loaded, a tree file will have to be read into the editor then loaded by way of the tree evaluation routines. The interpreter will need to be able to distinguish between the two cases. If we were able to convert text into ALOE program trees, then this would not be necessary.

3.3. Operating System Interface in AMPE

In defining the operating system interface in AMPE, we consider the interfaces of the ALOE editor and AML/X interpreter separately and determine how they need to be changed when the two are linked together. The needed changes to the operating system interfaces during integration are primarily due to input/output problems that arise. Therefore we focus on I/O problems. The problems of file I/O in AMPE involve redirecting some of the interpreter's I/O to the editor. The following diagram describes the various ways I/O could be done in AMPE:

I/O in AMPE

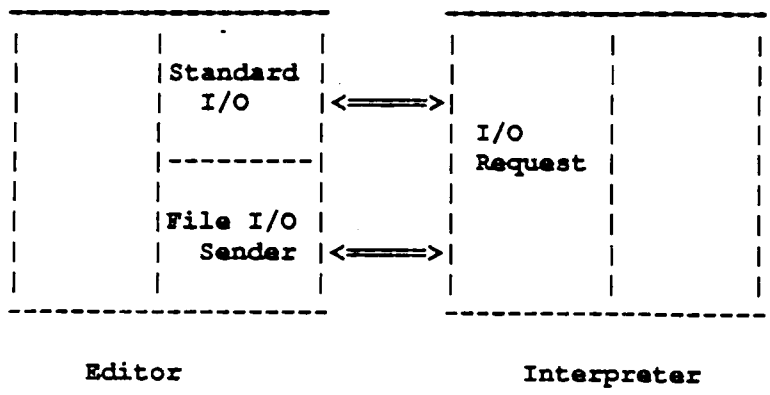


The editor can exchange information with the machine it is running on and with the interpreter. The interpreter can exchange information with the machine it is running on, with the editor, and, through the editor, with the machine the editor is running on.

All ALOE editors are generated with certain basic I/O routines such as those that load, save or unparse a file. The AML/X interpreter also has built-in routines for I/O. Interpreter I/O assumes its files are local. In AMPE, as mentioned earlier, this may not be the case. Therefore, the interpreter should have some way of determining whether a file is local or remote. Since this seems to be an issue for the designer of the interpreter, it is not dealt with here. In this thesis we assume that all interpreter I/O is directed to the editor. We have already discussed how program information reaches the interpreter. This information can be about an ALOE program tree or it can be simply AML/X text. The latter case is already an example of remote file I/O by the interpreter. In this section we will discuss in more detail how remote I/O is done.

The interpreter's I/O will need to be redirected as follows:

Remote Interpreter I/O



3.3.1. ALOE I/O

There are three kinds of editor I/O in AMPE:

1. I/O that is common to all ALOE editors - The I/O routines that load ALOE programs in tree form into the editor, store or unparse programs, etc. are provided by the ALOE kernel.
2. Communication with the interpreter - The ALOE editor must be able to send evaluation requests to the interpreter and replace standard I/O in the interpreter. Because the editor is screen-oriented, standard input will be through a command line, and standard output will be through a pop-up window.
3. I/O that is done by the editor on behalf of the interpreter - Since the interpreter may run on a machine different than the one where the editor runs and where the programmer is working, some interpreter I/O must be directed through the editor.

Since the I/O routines for a pure ALOE editor are already accounted for in the ALOE generator, we are not concerned with these routines. The remaining two kinds of editor I/O requires that editor I/O be augmented to include the following:

1. **I/O Server** - This server is defined by the I/O routines that it will permit the interpreter to execute remotely. In order to process I/O requests from the interpreter, we will need routines that execute the necessary system calls for the interpreter.

Communication with the interpreter occurs when the editor needs to have something evaluated and when it receives the results of the evaluation from the interpreter. Communication also occurs when the editor is doing I/O for the interpreter. The communication between the editor and interpreter will also be discussed in more detail later from the point of view of the interpreter.

The routines that allow the editor to communicate with the interpreter for evaluation purposes, should also include routines that allow the editor to do I/O for the interpreter. The last section discussed how to send program tree information. The sending of AML/X text and other data will be covered here. I/O that is requested by the interpreter can be of two types:

1. Standard I/O - Since ALOE is screen oriented, standard input will be through a command window. All output from the interpreter is directed back to the interpreter and is displayed in a pop-up window. The text is displayed just as it would be if someone were running the interpreter without the ALOE interface. The routines in the editor that handle standard I/O are written in ARL.
2. File I/O - The editor needs routines that will manage file I/O for the interpreter.

Changes to the operating system interface will occur when the editor does I/O on behalf of the interpreter. It is not clear how much the operating system interface needs to be changed. The interpreter will need to be able to do I/O on the editor's machine, but it is not clear how complete the interpreter's interface with the editor's machine needs to be.

3.3.2. Interpreter I/O

The interpreter needs to get information about the programmer's code from the editor in order to evaluate it. The interpreter also needs to send the output of the interpreter to the editor so that the programmer can see it. The interpreter needs to be able to do file I/O. Since the interpreter may run on a machine different than the one the editor is running on, I/O may have to be directed through the editor.

Interpreter I/O can be of three kinds:

1. **ALOE Program Information** - The interpreter needs a new parser in order to interpret information about ALOE program trees. The interpreter also needs to retain the original parser in order to be able to parse existing AML/X code in text form. This is necessary when executing the AML/X *load* command.
2. **Remote I/O** - This kind of I/O involves communication with the ALOE component. When input is needed from the user, the interpreter needs some way of indicating that input from the user is required, and getting that input. When output goes back to the user, it must be displayed in an output window. The interpreter needs to be able to indicate that a particular file needs to be opened or closed, or that a particular file needs to be written to or read from. The data being written or read by the interpreter will also have to be passed between the interpreter and editor.
3. **Local I/O** - This kind of I/O involves interaction between the interpreter and the operating system on which it is running.

As mentioned earlier, various communications protocols will need to be established between the editor and interpreter so that interpreter I/O will be possible. In AMPE we assume that all I/O files are remote, or local to the editor and programmer. Differentiating between local and remote files seems to be an issue for the interpreter and is not addressed in this thesis.

Interpreter I/O will have to be augmented to include the following:

1. **Parsers** - The interpreter will need to recognize and process evaluation requests.
2. **Remote I/O Routines** - The interpreter will also need routines that execute all other I/O requests.

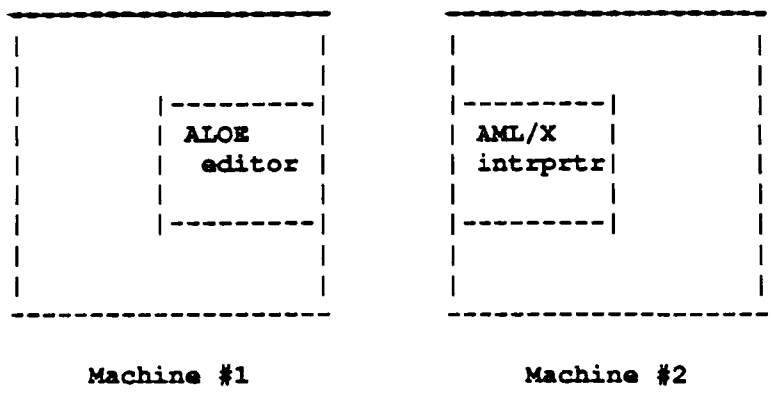
4. Design

AMPE consists of two main parts: the ALOE editor part and the AML/X interpreter part, each of which runs as a separate process. They use a communication protocol.

4.1. Communication Protocol

In AMPE the editor and interpreter are related in a *client-server* relationship. Either can be a client or server. When the editor needs something to be evaluated it is the client and the interpreter is the server. When the interpreter needs I/O, it is the client and the editor is the server.

AMPE While Editing



4.1.1. ALOE Component as Client

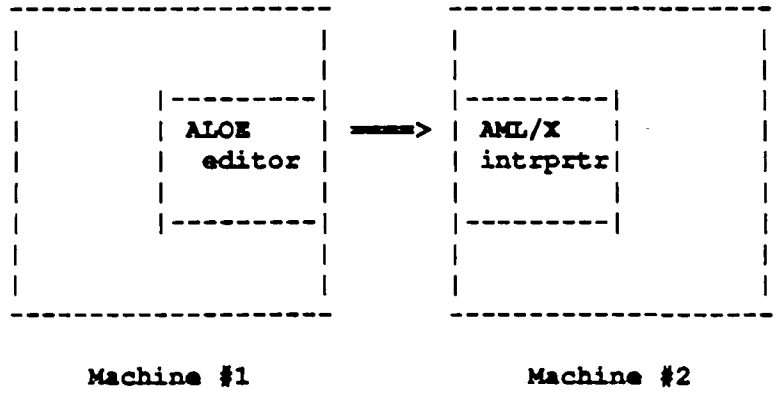
The editor can make the following request to the interpreter:

1. **evaluate** - Evaluate an ALOE program subtree.

Therefore, the interpreter first must indicate that the request is being made so that the interpreter can apply the correct parser to the interpreter input.

The editor is chiefly used as an interface to the interpreter. While the programmer is programming, the interpreter waits as a server for the editor to send an evaluation request.

Editor Makes an Evaluation Request



4.1.2. ALOE Component as Server

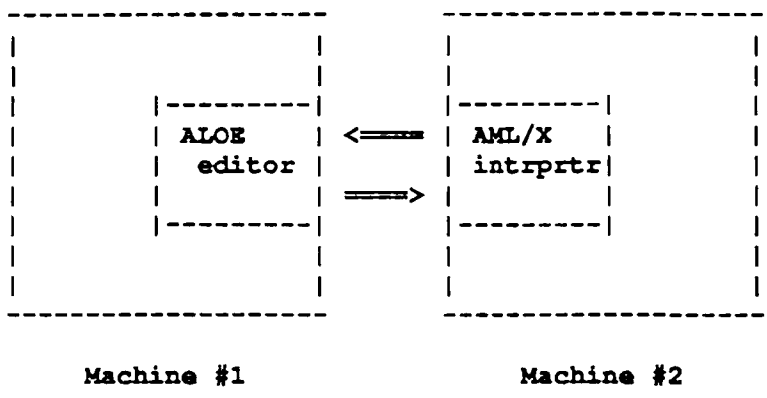
Once the interpreter is evaluating AML/X code, whether it is in tree or text form, it may become necessary to do I/O. While the interpreter is evaluating, the editor is waiting for interpreter output. At this point, the editor can become a server for an interpreter I/O request:

1. **Input** - After the interpreter has indicated an input request, it then specifies the source. If standard input is specified, the user is prompted and the the input obtained. If a request is made for a file to be opened or for a file to be read, then the appropriate action is taken.
2. **Output** - Output requests are handled in much the same way as the input requests.

After all interpreter I/O is finished, the editor is informed and the interpreter goes back to waiting for an

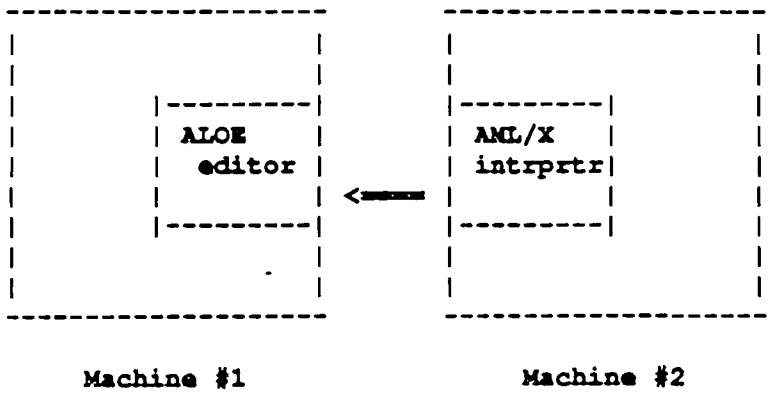
evaluation request.

Interpreter Makes I/O Requests



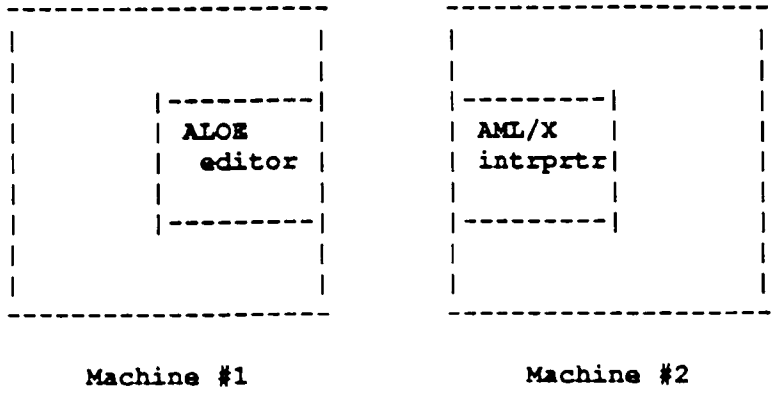
After sending an evaluation request, the editor waits as a server for any possible requests from the interpreter. The interpreter finishes evaluation and signals that it is finished.

Interpreter Signals End of Evaluation



The editor now displays the result of the evaluation in a pop-up window. We are now back in the original editing state.

Original State



4.2. The ALOE Component

This section will discuss the design and implementation of the ALOE component. The work on the ALOE component can be divided into three parts:

1. **Structure Editor** - This part includes all the generic ALOE editor commands.
2. **Evaluation Routines** - The routines responsible for sending messages about the AML/X code to the interpreter. This part also includes other tree traversal routines that return information about the ALOE program trees.
3. **Remote I/O Routines** - These routines are responsible for replacing the standard I/O of the interpreter and handling file I/O for the interpreter.

4.2.1. Structure Editor

In order to generate an ALOE editor, we need to provide a language description of AML/X. ALOEGEN is used to develop the syntax of AML/X which is used by the ALOE component of AMPE. An abstract syntax is given which is used to generate the attributed syntax tree used by ALOE editors. Concrete syntax is given in the form of unparsing schemes, which dictate how the program tree is displayed to the programmer and what it looks like when printed as a text file. Most of the work done here is standard for any ALOE editor.

4.2.2. Evaluation Routines

The main evaluation routine is the *evaluate* extended command which causes the evaluation of ALOE trees by the interpreter. ARL routines have been used for traversal of the program tree, accessing different fields of the tree, and for display management. C functions have been used for implementing the interprocess communication. DBGEN is used for specifying what routines are called to execute what *extended* command. It is also used to specify initialization routines to be executed when an editing session is being started, and clean-up routines to be executed when the editing session is being ended. These routines are used to initialize a socket connection between the editor and interpreter, and used to disconnect the editor and interpreter. This causes the interpreter to terminate itself.

When evaluating trees, we will need routines for basic tree traversal problems. Therefore we have several **tree traversal** routines. For example, when setting breakpoints during debugging, it would be useful to have a command that traverses the program tree and returns line numbers. The AML/X interpreter expects text and determines line numbers when parsing so that it can return line numbers in error messages for example. When evaluating with trees, we must supply these line and character numbers ourselves.

The following tree traversal routines will be needed for a variety of reasons:

1. **Intonode** - this routine locates the node of a program corresponding to a given line number. Error messages returned from the interpreter may contain references to line numbers. Such references to line numbers require that the user of the AMPE, who will be editing a tree structure, to be able to locate a given line number.
2. **nodetoln** - this routine returns the line number given a node of a program tree. In order to use existing AML/X debugging routines, it is necessary to determine line numbers as when setting break points. Given a node in a tree structure, it is not always immediately obvious what the corresponding line number is.
3. **findameta** - this routine determines if a meta node exists within a tree for which evaluation has been requested. This is to prevent attempted evaluation of an incomplete program tree that still has *meta nodes* in it.

While the first two routines are provided as *extended* commands for direct use by the programmer, they are also called form higher level functions. As mentioned earlier, *nodetoln* can be automatically called when an *extended* command is setting a breakpoint. *Nodetoln* is also called from the *evaluation* routine. The AML/X interpreter requires linenumber information when parsing so that when errors occur it can return line numbers. *Nodetoln* is used in sending this information to the interpreter when the program information is sent. *Lntonode* can also be called automatically when the interpreter returns an error message so that the programmer's cursor is automatically moved to the correct line number. *Findameta* is also called by the *evaluation* routine.

In implementing *lntonode* and *nodetoln* several generic routines are used for counting line numbers. Special routines need to be written for language constructs with unique unparse schemes, such as *Conditionals* and *Class* definitions.

In the ALOE generator, the ARL language makes it easy to access the fields of ALOE tree nodes and to manipulate these trees in general. The AML/X interpreter has its own tree building and tree manipulation routines. Since ARL routines work on ALOE syntax trees and the interpreter has its own routines to work on its own data structures, we need to have ARL routines cause the proper creation and management of the interpreter's data structures by causing the proper interpreter routines to be called in the proper order. Using these routines we implement the **evaluate** command.

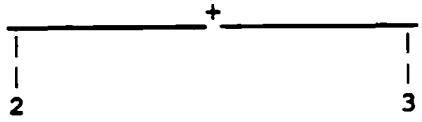
For example, consider the AML/X expression:

```
2 + 3;
```

The AML/X interpreter parses this as:

```
Addition Operator +
Integer           3
Integer           2
```

AML/X uses a YACC generated parser so the tree is represented as a stack. The original AML/X interpreter recursively parses the two children of the "+" node, pushes the result of each recursive parse on the stack then pushes the "+" operator. The ALOE tree representing the same expression would look like this:

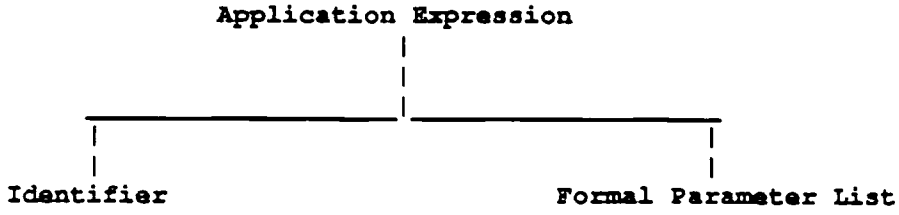


The language description of AML/X used to generate the ALOE editor uses a *flattened* grammer so that ALOE uses tighter trees that are easier for programmers to write. For example, a "2" can be directly specified as an operand of the addition operator. The programmer does not have to specify the operator as first an expression, then integer, then the number "2".

When evaluating the ALOE tree, a tree traversal causes the first operand to be parsed. It is determined to be the integer "2" so the appropriate routine from the original parser is called to push a "2" on the parse stack. If the interpreter were imbedded, the routine that pushes integers would be called directly. With the interpreter in another process, a message indicating that an integer should be pushed is sent followed by the integer to be pushed. Likewise, "3" is pushed on the stack and finally the "+"

operator is pushed. The result is a parse stack, identical to the one above, that can be evaluated by the evaluation routine of the AML/X interpreter.

Language commands such as *resume* can be easily implemented within the general evaluation scheme. AML/X language commands are always *application_expressions*, which are represented as the ALOE tree:



The *identifier* part of the *application_expression* can be supplied implicitly by associating an AML/X command with a particular *extended* command.

The *formal_parameter_list* part can be specified by prompting the programmer for input. Since we are not constructing program trees, the input is in text form and so there are restrictions on what the programmer can pass as parameters. While AML/X allows an arbitrary expression to be passed as a *formal_parameter*, we do not allow arbitrary expressions to be entered. The reason is that we would have to convert the AML/X text into an ALOE tree. This problem will not be addressed in this thesis. Therefore input will be limited to a particular type of literal depending on the particular AML/X language command. For example, the *extended* command that executes the AML/X *load* command will prompt for a character string representing a file name.

When possible, these arguments should be passed automatically. An example of this would be when setting breakpoints. Suppose the programmer wants to set a breakpoint at the current cursor position. The interpreter needs a line number in order to set the breakpoint. Such line numbers can be automatically provided through routines that automatically traverse the program tree.

4.2.3. Remote I/O Routines

Routines in the editor that do I/O for the interpreter can be of two types:

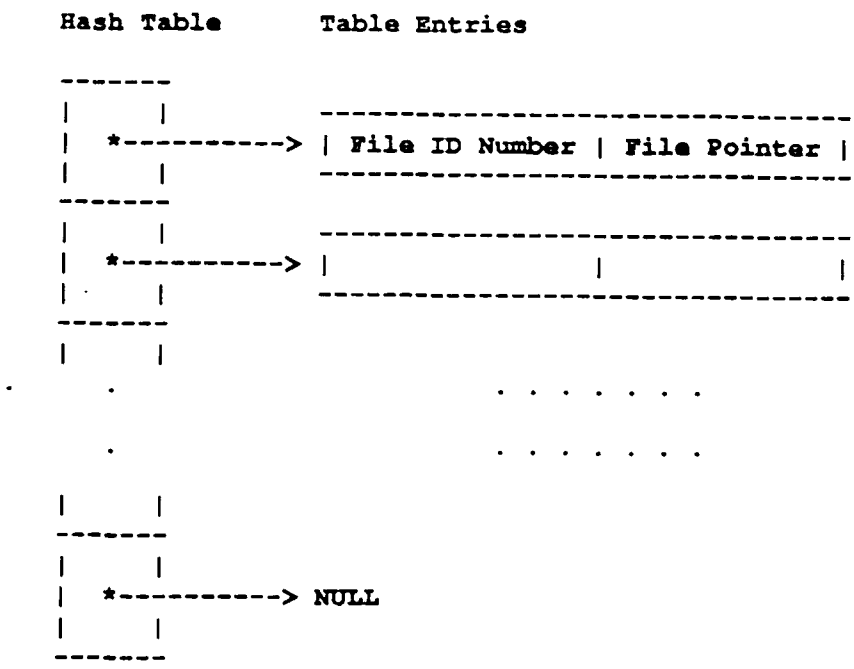
1. **Standard I/O Routines** - These routines display information to the editor, and get information from the user. These routines are written in ARL, which is used most of the time when implementing a normal ALOE editor. ARL contains many functions for displaying information and reading it from the user.
2. **File I/O Routines** - These routines interact with the operating system on which the editor is running. Since these routines do I/O for the interpreter, these routines are written in C.

The standard I/O routines are restricted to routines that display text to the user of AMPE and read text input from the user. Text output is displayed in a pop-up window and input is prompted for and taken from a command line. The file I/O routines allow the interpreter to work remotely on files. Therefore, the implementation of the file I/O routines are more involved. In this section we will first discuss the data structures used for managing the file I/O data and then the actual routines themselves.

Each file is identified by a *file identification number*. In AMPE, we simply use the file descriptor. File pointers are stored, along with their corresponding file identification number, in a hash table. The hash

table itself is an array of pointers and looks like this:

File Pointer Hash Table



The remote operations on files that are currently allowed are:

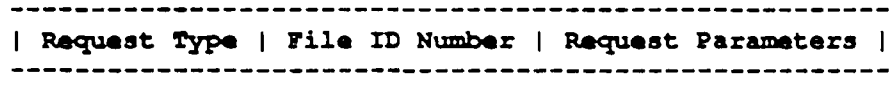
Remote I/O Routines

1. **open** - opens a file.
2. **read** - reads from a file.
3. **write** - writes to a file.
4. **close** - closes a file.
5. **flush** - causes input buffers to be emptied and output buffers to be written.

When the interpreter wants to execute one of these remote routines, it sends a remote I/O request.

The format of a remote I/O request is:

Format for Remote I/O Request



Open requests omit the file ID number.

The remote use of a file by the interpreter will occur as follows. When the editor receives an **open** request, the editor executes the *fopen* routine in the C language. The *fopen* routine is executed with the parameters given by the interpreter. If a file is successfully opened, the file pointer and corresponding file identification number is stored in the hash table at the index determined by hashing on the file identification number. This file ID can be any integer determined by the editor. Here, we have chosen

simply to use the file descriptor of the opened file. When a **read** request is received, the editor hashes on the given file ID number and executes *fread* on that file with the parameters supplied by the interpreter. The results of the *fread* command are then sent back to the interpreter. The **write** command is executed in a similar manner. The **close** command causes the *fclose* command to be executed and the appropriate entry in the hash table to be removed. The **seek** command executes *fseek*.

In a similar way, it is also possible to add routines that do seeks on a file by executing *fseek* or check if *end-of-file* has been reached by executing *feof*, and so on. It would also be possible to allow other system calls to be executed remotely, but it is not clear which other ones would be useful.

4.3. Interpreter Component

The interpreter component is responsible for code evaluation in AMPE. The evaluation routines, which naturally constitute the vast majority of the code in the interpreter has been left in its original form. However, the interpreter had to be modified in three areas:

1. A second parser was written to read tree information from the editor.
2. File I/O was directed to and from a socket.
3. The main *read-print-eval* loop communicates with the editor in order to stay synchronized.

The main *read-eval-print* loop now runs as follows:

1. **Read** - Determine from the editor which parser to run on the coming input.
2. **Eval** - Run the appropriate parser and evaluate. During evaluation, do the needed I/O.
3. **Signal End** - Signal the editor that the evaluation is complete and a new request can be sent.

4.3.1. Parsers

The interpreter will read two kinds of input:

1. **Tree Information** - The interpreter needs to be able to receive information about ALOE program trees so that it can do the appropriate evaluation. A "tree parser" was written for this purpose.
2. **Text** - The interpreter needs to read text data while doing evaluations. It also needs to read and parse AML/X text when evaluating code in text form.

This section is concerned with the tree information input and the *tree parser* needed to interpret ALOE tree information. The AML/X interpreter already comes with a text parser. All other text I/O will be discussed in the following section.

The results of parsing with the tree parser and original text parser are the same. In either case the same tree building routines, etc. are called in the same order. In fact, the tree parser was derived by observing the flow of control in the text parser.

The AML/X interpreter uses a *YACC* generated parser. This **text parser** runs as a state machine in which one state is chiefly responsible for the actual construction of the parse tree. In this state is a large C language *switch* statement (similar to a *case* statement in Pascal). The parse tree is formed as control repeatedly enters various cases in this switch statement. The following simplified example, written in pseudo-code, illustrates how this works.

Block Text

```
Block Begin
     Declaration
     Declaration
     Statement
     Statement
     Statement
Block End
```

Text Block Parsing Code

```
   .              . . .
   .              . . .
case i:    initialize block
case j:    close block
   .              . . .
   .              . . .
case m:    parse declaration
case n:    parse statement
   .              . . .
   .              . . .
```

The above text would cause control in the interpreter to go to case *i* where a block would be initialized, then case *m* twice where each declaration would be accounted for, and then *n* three times, where each of the three statements would be parsed. Finally, control would go to case *j* and the block is closed.

When receiving program tree information, the **tree parser** causes the same sequence of routines to be called. The tree parser, however, is structured differently. The tree parser is a set of mutually recursive routines each of which handle a particular language construct. In the above example, there would be a routine that specifically handles *blocks*. The editor would send tree information about a block in the following format.

Tree Block Information

```
-----------------------------------------
| Block | Declaration List | Statement List |
-----------------------------------------
```

The tree parser routine to handle blocks looks like this:

Tree Block Parser

```
Function Parse_Block
     Begin
          initialize block
          loop
               parse declaration
          end loop
          loop
               parse statement
          end loop
          close block
     End
```

The routines that parse statements might themselves call the routine, *parse_block*.

4.3.2. Interpreter I/O

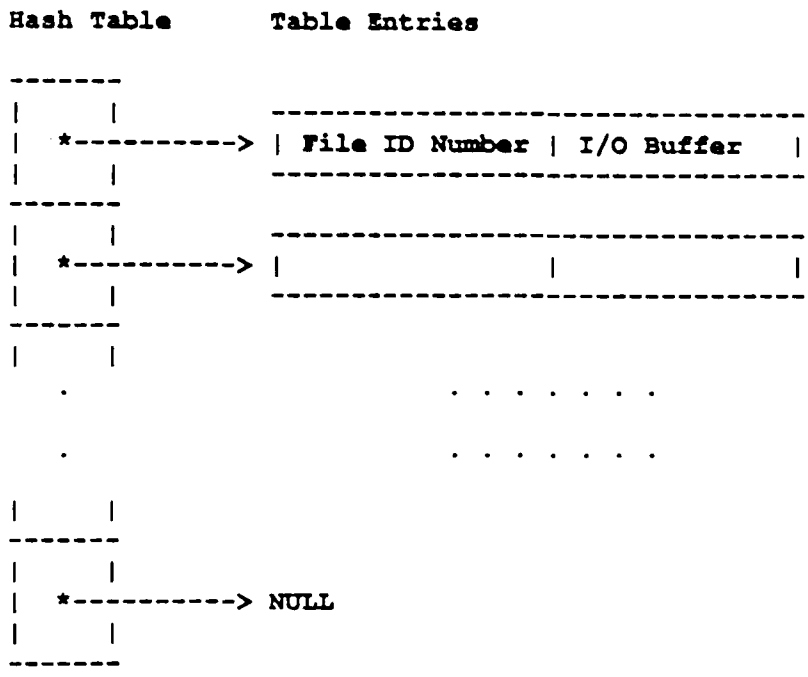
In AMPE we have redirected the interpreter's I/O from the operating system to the editor. In order to do this, we needed to replace some standard I/O calls in the interpreter's code with calls to routines that manage I/O with the editor. The replaced routines are the C language routines:

Replaced C Language Routines

1. **fopen** - opens a file.
2. **fread** - reads from a file.
3. **fwrite** - writes to a file.
4. **fclose** - closes a file.
5. **fflush** - empties input buffers and writes output buffers.

As with the corresponding routines in the editor component, hash tables are needed here. These hash tables are used to store I/O buffers between the editor and interpreter. One hash table is used to buffer interpreter input. The other is used to buffer output.

I/O Buffer Hash Table



The new **open** routine sends a request to the editor to open a file. It then receives the result of the open request. If the request is successful then an input and/or output buffer is allocated for that file, depending on the *read-write* permission for that file. The result of a successful open request is the *file identification number* which will identify the opened file as long as it remains open. The interpreter maintains a dummy file pointer and stores the file id number where the file descriptor would have been stored, had the original call to *fopen* been made. The *read-write* permission in this dummy file pointer is also set so that the interpreter would not have to make a request to the editor for this information when it can easily be stored locally. Another reason for storing this information is that in the interpreter's code, there is a macro defined that checks this *read-write* permission. The use of the dummy pointer in general allows us to redirect I/O without rewriting the data structures that manage I/O for AML/X. It also allows us

to keep some information about our files local, while at the same time allowing I/O to be done remotely.

If a **read** request is made with a particular file ID number, then the file ID number is hashed and the appropriate input buffer is checked. If possible, contents from the buffer are returned, otherwise a *read* request is sent to the editor and the buffer is filled. **Write** works similarly. A request is made with a file ID number. This number determines the output buffer into which data is written. Data is sent to the interpreter whenever the output buffer fills up. **Close** forces the contents of a file's output buffer to be sent to the editor and written out. It also removes all of that file's I/O buffers form the hash tables. **Flush** forces output buffers to be written, and empties input buffers.

5. Example

This section gives a small sample session with AMPE to show how it can be used. As illustrated in section 2, an AML/X program is created by filling in program templates. To further illustrate how AMPE works, we continue the example started in section 2.

Templates for Generic Declaration

```
$label $declarator

$stat_decl


Window:root Node:META          Class:label
>
```

Here we have the templates for the components of a generic program *declaration*. The underlining of *$label* indicates that that is the current cursor position. In order to create a *subroutine* declaration we move the cursor to *$declarator* and execute the command that fills in the templates for a *subroutine*. We can also move the cursor back to *$label* and enter a name for our subroutine.

Subroutine Declaration Template

```
incr: STATIC SUBR (formal arg)
    $declaration;
    $statement;
END;

$stat_decl


Window:root Node:META          Class:formal arg
>
```

By continuing to fill in templates we create a subroutine declaration.

Subroutine Declaration

```
incr: STATIC SUBR (n)
    return (n + 1);
END;

$stat_decl


Window:root Node:IDENTIFIER   Class:expression
>
```

This subroutine takes as input an integer and returns the sum of that integer plus one. In order to evaluate this subroutine declaration, we cover the entire declaration with the cursor and execute the *evaluate* extended command. To execute this subroutine, we need to make a call to this subroutine. This is done by creating an *application* expression and evaluating it. This is the same as making a *function* call.

Evaluation of Subroutine Call

```
incr: STATIC SUBR (n)
    return (n + 1);
END;

incr(x);

$stat_decl


Window:root Node:APPLICATION_EXPR   Class:stat_decl
>
```

The result of the evaluation is displayed in a pop-up window. In this case, the result is an error message indicating an arithmetic error due to the unbound variable *x*. The message indicates the line number at which the error occurred. The *character number* in the message actually refers to a *leaf number*, since AMPE edits tree structures not real text.

Error Message

```
AML:
*** EXCEPTION EXCP_NOTNUM:
   Number (INT, LONG_INT, REAL, or LONG_REAL) expected. ***
    Operator: +
    Operands: < UNBOUND, 1 >
    File:
    Line: 2
    Char: 2
    Data: UNBOUND

Window: Interpreter Output          Region: Evaluation
```

At this point, we have a break in execution which allows us to examine the values of various variables. We could also execute the AML/X debugging commands to set explicit break points. The *Intonode* routine (section 4.2.2) moves the cursor to the appropriate line in the displayed program.

Cursor at Source of Error

```
incr: STATIC SUBR (n)
    return (a + 1);
END;

incr(x);

$stat_decl


Window:root Node:IDENTIFIER   Class:expression
>
```

We can resume execution using the AML/X *resume* command. After substituting a *6* for *x* in the call to *incr*, we get the following pop-up window.

Final Evaluation

```
AML:
7

Window: Interpreter Output          Region: Evaluation
```

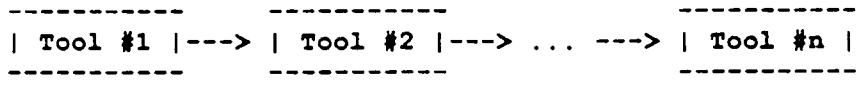
6. Relation to Other Models of Tool Integration

David Garlan [4] characterizes approaches to tool integration as variations on two models, the "sequential" and "concurrent". AMPE has features in common with both models of tool integration. As in the *sequential* model, each tool works on its own separate data structures. As in the *concurrent* model, the tools run at the same time.

6.1. "Sequential" Models of Tool Integration

In the "sequential" model, tools communicate sequentially through well defined interfaces. An example of this kind of system would be the Unix operating system, which provides programmers with a set of programming tools. These tools can be linked using Unix pipes, which provide a way of channeling the output of one tool into another. Among the advantages of this approach is the modularity of the components. Each tool maintains data representations appropriate for itself. And the implementor of each tool can work without concern for the implementation details of the other tools. Conversely, the implementor need not worry about outside influences on his code. This modularity can make it easy to modify existing tools or add new tools to the system. One of the disadvantages is that the data representations used by one tool may be inappropriate for another. Therefore, some transformation must take place. This may be difficult or expensive to do.

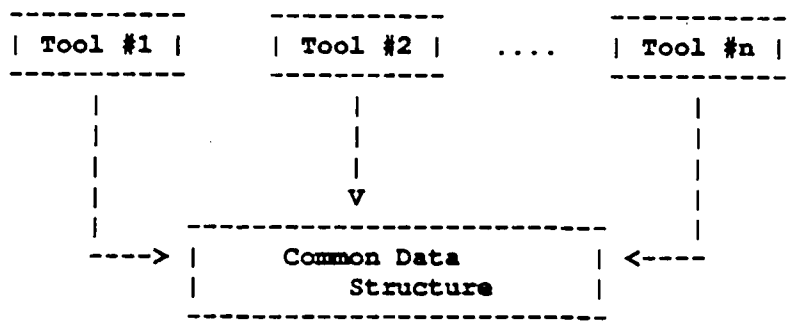
Sequential Tool Integration



6.2. "Concurrent" Models of Tool Integration

In the "concurrent" model, tools have shared access to a common database of programming objects. As is the case with ALOE editors, the data representation is typically an attributed abstract syntax tree. The advantages of this approach include the ability to integrate tools. For example, in an ALOE editor, editing commands can trigger semantic checking automatically when appropriate. Another advantage is that data does not need to be replicated for different tools as in the "sequential" model. The disadvantage is the lack of modularity in this model. It is difficult to work on separate tools without affecting other tools and it is difficult to add new ones.

Concurrent Tool Integration



6.3. Examples of Tools Integration

The limitations of both models have lead to research into other intermediate forms. An integration mechanism called Odin [2] has been used to integrate families of existing Fortran and C programming tools. Garlan [4] has proposed a model of tool integration that provides tighter integration, but is not as applicable to diverse collections of existing tools. Snodgrass [11] takes a similar approach. Marvel [1] represents a knowledged-based approach to tool integration. Marvel, like Odin, works with existing programming tools. The RPDE^3[6] allows fragments of tools stored in a library to be recombined.

A system that is similar to AMPE in structure is David Garlan's *DemoGen* system [3]. The ability to run an ALOE editor and another program in separate processes, communicating with each other, has been demonstrated by this system, which generates demonstration programs for software systems. Such a demonstration program has run interactively with the *Gandalf* environment generation system [5], of which the ALOE editor generator (ALOEGEN) is a component.

7. Conclusion

Integrating an ALOE editor to the AML/X interpreter demonstrates the restrictions of the "concurrent" model of tool integration, and shows how an ALOE editor can be extended in a limited way to take advantage of runtime support running on another process. Further research that can be done on AMPE can be divided in two parts:

1. **Further Work on AMPE Itself** - There is more work to be done on just having the ALOE editor and AML/X interpreter working together.
2. **Generalization of the AMPE Model** - AMPE consists of two parts: an editor and an interpreter. It is not clear how other tools can be added.

7.1. Further Work on AMPE

Most of the future work to be done on AMPE involves the use of text files.

1. **ALOE I/O with Text Files** - We would like to be able to load program text into ALOE for editing.
2. **Interpreter I/O with Text Files** - The interpreter needs to be able to differentiate local and remote I/O files.

7.1.1. ALOE File I/O

About the problem of reading text into AMPE, the solution consistent with the other work here would be to make use of the AML/X interpreter to do the parsing first and then have the tree transformed into one usable by the ALOE editor. Therefore, the *edit* command would have to be defined that retrieves a given file, and calls the interpreter's parser to generate an initial tree, which could then be transformed and loaded into the ALOE editor.

There will be problems in reading program text files into the ALOE editor and unparsing them later. One problem is that the way a programmer writes his code probably will not match the unparsing scheme of the editor. Therefore, if a program is loaded into the ALOE editor and unloaded, using the editor's unparsing scheme, the appearance of the program can be completely changed. There is an additional problem of how to handle comments in parsing the text program. Usually they do not become part of the parse tree. But clearly the comments will become useless if they do not. In order to handle comments the AML/X parser itself would have to be modified. Since comments can appear anywhere in the program, it is not clear how they can be stored. And when unparsing occurs, we would like the comments to be returned to their intended locations.

One approach to this problem would be to include comments as part of the syntax of AML/X. While this approach could be an acceptable solution for a programmer working on a program tree, it would put severe restrictions on the writer of text files, and would require that existing text programs be recommented.

If this problem is solved, then we will no longer need a text parser, since we will no longer need to send AML/X text to the interpreter. We can first transform program text into an ALOE tree and then do a tree evaluation on it. This would simplify the interface between the editor and interpreter.

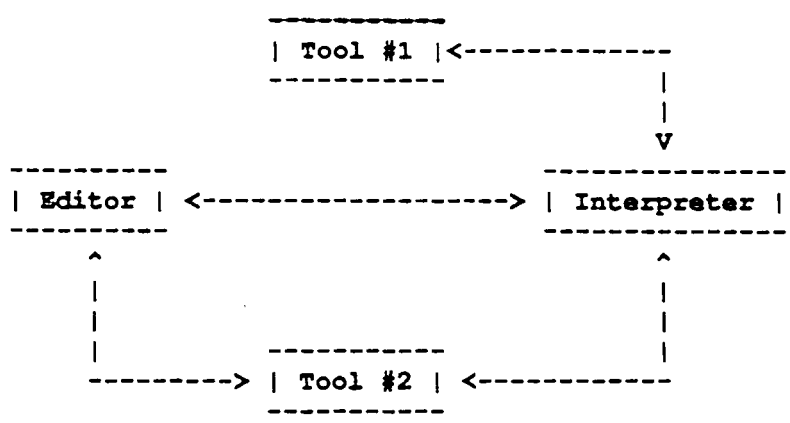
7.1.2. Interpreter File I/O

It is probable that an AML/X program may need to use some files that are local to itself and some that are local to the editor. In that case the interpreter must be able to differentiate requests for local or remote I/O. This kind of differentiation seems to be a language issue and so has not been addressed here. We have assumed that when the AML/X interpreter makes an I/O request, that the desired file is local to the editor and programmer.

7.2. Generalization of AMPE Model

AMPE functions as two programs, an editor and interpreter, running on different processes. In order to generalize this model, we will need to be able to add new programming tools. That means that we will need a more general way of generating interfaces between the tools.

Generalized AMPE



There is another problem that will arise when adding new tools. The editor and interpreter in AMPE run as coroutines. Although they run in separate processes they do not run concurrently. When adding more tools we may want to have some tools running concurrently either for efficiency reasons or just because of the natures of the tools themselves. In this case the interfaces between the tools will become more sophisticated. One approach to creating generalized interfaces between programming tools would be to try to use a "Marvel-like"[8] *envelope*. Marvel uses *envelopes* to describe a tool to the system so that it can use it without altering it. An *envelope* is basically a shell-script that can redirect input and catch output.

References

1. N. S. Barghouti and G. E. Kaiser, "Implementation of a Knowledge-Based Programming Environment", Twenty-first Annual Hawaii International Conference on System Sciences, vol. II, January 1988, pp. 54-63.

2. G. Clemm and L. Osterweil, "A Mechanism for Environment Integration", Technical Report CU-CS-323-86, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1986.

3. D. B. Garlan, "DemoGen: A Development System for Demonstration Programs", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, March 1982.

4. D. B. Garlan, "Views for Tools in Integrated Environments," in: R. Conradi, T. M. Didriksen and D. H. Wanvik Eds., Advanced Programming Environments, Lecture Notes in Computer Science 244 (Springer-Verlag, Berlin 1986) pp. 314-343.

5. A. N. Haberman and D. Notkin, "Gandalf: Software Development Environments", IEEE Transactions on Software Engineering, vol. SE-12 No. 12, December 1986, pp. 1117 - 1127.

6. William Harrison, "RPDE^3: A Framework for Integrating Tool Fragments", IEEE Software, Nov. 1987, pp. 46-56.

7. G. E. Kaiser and C. W. Krueger, "Using The New Gandalf System (A Tutorial)", Technical Report CMU-CS-86-146, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, August 1986.

8. Gail E. Kaiser, Naser S. Barghouti, Peter Feiler, and Robert Schwanke "Database Support for Knowledge-Based Engineering Environments," IEEE Expert, Summer 1988, pp. 18-32.

9. Charles W. Krueger, A. N. Habermann, "The GANDALF Editor Generator Reference Manuals" in "The GANDALF System Reference Manuals," Technical Report, Computer Science Department, Carnegie-Mellon University, 1986.

10. L. R. Nackman, M. A. Lavin, R. H. Taylor, and W. C. Dietrich Jr., "AML/X User's Manual," IBM Research Report RA 175, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, (1986)

11. R. Snodgrass and K. Shannon, "Supporting Flexible and Efficient Tool Integration", in: R. Conradi, T. M. Didriksen and D. H. Wanvik Eds., Advanced Programming Environments, Lecture Notes in Computer Science 244 (Springer-Verlag, Berlin 1986) pp. 290-313.

12. B. J. Staudt and V. Ambriola, "The ALOE Action Routine Language Manual," in "The GANDALF System Reference Manuals," Technical Report, Computer Science Department, Carnegie-Mellon University, 1986.

Appendix

I. AMPE User's Manual

I.1. Introduction

AMPE (A Manufacturing Programming Environment) is a programming environment for the robotics programming language, AML/X. AMPE runs in two processes. One process runs a structure editor. In the other process runs the AML/X interpreter. With the editor, AMPE supports the development of syntactically correct code. With the interpreter, AMPE provides run-time support for the programmer. The user of AMPE can view the editor as just an interface to the interpreter.

Model of AMPE

Programs are developed in a structure editor by filling in program templates. The user of AMPE therefore edits program trees, not text. Using the structure editor, AMPE manipulates program trees and eliminates some of the more tedious aspects of programming, such as matching *begin* and *end* statements and placing semi-colons correctly. It is impossible, using this kind of editor, to create program trees that are syntactically incorrect. To illustrate how a structure editor works, we show how a small arithmetic expression is created. The interpreter expects to evaluate *statements* or *declarations*. The template for a *statement* or *declaration* is represented by *$stat_decl*. The components of templates that get filled in are called meta nodes. Meta node names are indicated by *$*. Therefore the first template that the programmer will see is:

Template for Statement or Declaration

```
$stat_decl;
```

The required semicolon is automatically placed. A statement can be a simple arithmetic expression so we can insert the template for the addition operator.

Template for Arithmetic Expression

```
$expression + $expression;
```

This template, in turn, requires two operands to be specified. These operands must be AML/X *expressions*. Expressions can be integers so that we can insert integers into these templates.

Completed Arithmetic Expression

```
4 + 7;
```

AMPE also allows program trees to be evaluated by an interpreter during an editing session. The interpreter itself provides debugging facilities. Evaluation of the above arithmetic expression returns the interpreter prompt and the result of evaluation.

Result of Evaluation

```
AML:
11
```

In the next section, we will describe how to generate and edit AML/X programs. After that we will describe some of the run-time support offered by the AML/X interpreter.

I.2. Editing

The structure editor component of AMPE has been generated by the *Gandalf* programming environment generation system [1]. Such editors are known as ALOE (A Language Oriented Editor) editors. All ALOE editors have at least two kinds of commands.

1. **Editing** - Editing commands include commands such as those that delete, review and move previously created program information.
2. **Constructive** - Constructive commands create program information.

The ALOE commands are executed with *EMACS-style* keystrokes and should seem familiar to user's of that text editor. The rest of this section describes the basic ALOE editing commands. The purpose of this section is to explain only the most basic commands to the novice user. These descriptions are largely excerpts from the ALOE User's Manual [2], which the user of AMPE is referred to for more detailed and complete explanations of the ALOE commands.

Before running AMPE, it is necessary to have an *.aloeprofile* file in your UNIX home directory. This file should contain:

```
all:
novice yes
```

This simply identifies you to be a novice user to the ALOE editor.

Entering and Exiting AMPE

Files edited by an ALOE are stored as UNIX data files in "tree format". The extensions for tree files are ".tr". For example, an ALOE program called TEST is stored in the file TEST.tr. In order to run AMPE, the interpreter component must first be running. The user of AMPE may run the interpreter in a background process. This is done by typing:

```
amlx &
```

to the UNIX shell. The basic method of invoking the ALOE component is by typing "ampe <program name>" to the UNIX command line. If the <program name> file does not exist then ALOE creates a new file. In order to edit the program, TEST.tr, you would type:

```
ampe TEST.tr
```

to the UNIX shell.

Two windows appear on the screen. The bottom line of a window, called the status line, is highlighted and contains the window name and information about the status of the window. In this case the top window is the program (or root window) and contains the template for an AML/X program. The bottom

window is the help window and contains the list of valid operators for the highlighted node in the program window. The bottom two lines on the screen are the command and message lines. All keyboard input that you type appears on these lines. All messages to you from the ALOE component are also displayed here.

There are two ways to exit from AMPE. If you enter *control-X control-F* (^X^F) AMPE writes out the current tree into the <program name>.tr file, while entering a ^C exits without writing out the file and all changes made during the editing session are ignored. If you have made changes during an editing session and you type a ^C to abort the session, AMPE asks for confirmation:

```
The program has been changed.  Exit anyway? [no]:
```

The "no" in brackets indicates the default value. Typing a *Carriage Return* (<CR>) selects the default value and in this case AMPE would not exit. Entering a *yes* <CR> terminates the session. Exiting from AMPE in either of these ways causes the interpreter also to exit.

Cursor Motion

The AMPE cursor corresponds to a single node in the program tree, called the current node. AMPE displays the current node by highlighting all of the text associated with it and its subtree. There are two basic commands for cursor motion:

Motion through the tree

These commands are all explicit tree motion commands.

1. Move the cursor **OUT** to the father of the current node (ESC-P).
2. Move the cursor **IN** to the left-most son of the current node (ESC-N).
3. Move the cursor **PREVIOUS** to the left sibling of the current node (ESC-B).
4. Move the cursor **NEXT** to the right sibling of the current node (ESC-F).
5. Move the cursor **HOME** to the root of the tree or root of the current window (ESC-?).

Motion through the text

These commands are implicit tree motion commands. Although the cursor always points to a node in the tree, these commands move the cursor relative to the displayed text.

1. Move the cursor to the next line (^N).
2. Move the cursor to the previous line (^P).
3. Move the cursor forward (^F).
4. Move the cursor backward (^B).
5. Move the cursor to the beginning of the line (^A).
6. Move the cursor to the the end of the line (^E).

When the cursor is moved onto a meta node the help window displays the list of legal operators for that node. The operators are the commands that are used to fill in meta nodes.

Scrolling

Text in the current window can be scrolled (the current window has the cursor highlighted in it). ^V scrolls the text forward one window full, and ESC-V scrolls back one window full. ^Z scrolls one line forward and ESC-Z one line back.

Construction of ALOE Trees

This section contains an editing session with AMPE. At this point, we assume that AMPE has just been entered and editing will begin on a new file.

As in the example in the introduction, the first meta node to be filled in is the *$stat_decl* meta node. The underlining of *$stat_decl* indicates the current cursor position.

```
$stat_decl;
```

In the help window you see all of the valid operators that can be applied at this point to fill in this meta node. Since there are more possible operators than fit into the help window, a message appears on the bottom of the screen to assist you in scrolling through the help window. Construct a *while_collect* template by typing *while_collect* on the editor command line and typing <CR>. Another way of doing is to type enough of the command to uniquely identify it and then pressing the escape key (<esc>) to complete the command. A *while collect* construction is a version of the *while* loop that returns an aggregate of the results of evaluating the body of the *while* loop. The cursor is moved to the first offspring of the *while_collect* template, which is the expression that determines whether the loop is entered. Since AML/X programs are simply lists of statements and declarations, a new *stat_decl* meta node is create automatically.

```
WHILE $expression
    DO COLLECT $expression;

$stat_decl
```

To enter a *less than* expression type *lt* <CR>. The *less than* operator takes two operands which are in turn expressions. The first operand will be a *expr_pplus* expression which returns a value and then increments itself. Some of the operators listed in the help window have values inside of parenthesis. These values are called *synonyms*. Either the entire operator name or its synonym can be entered. Therefore, we can either type *expr_pplus* or its synonym, *expr++*.

```
WHILE $expression++ LT $expression
    DO COLLECT $expression;

$stat_decl
```

To enter a simple variable *i*, first enter *identifier*. AMPE then prompts with *value:* to which we respond with *i*. The cursor now moves to the next meta node.

```
WHILE i++ LT $expression
    DO COLLECT $expression;

$stat_decl
```

We can insert another variable, *n*, here just as we entered *i* before. But many of the terminal nodes in the editor have lexical routines associated with them. If you enter an *n* now, the editor checks all of the

lexical routines to see if it is a valid entry for them. Here the lexical routine for identifiers accepts *n*.

```
WHILE i++ LT n
    DO COLLECT $expression;

$stat_decl
```

We enter a multiplication expression by typing *multiply* or its synonym, *. The integer, *3*, can be entered directly because the lexical routine for integers will recognize it. An identifier can be entered as before to give us:

```
WHILE i++ LT n
    DO COLLECT 3 * i;

$stat_decl;
```

When evaluated and when *i* is initialized to 0, this loop will return the first *n* integral multiples of 3.

Modification of Existing ALOE Trees

This section continues with the example from the previous section. Suppose we want the first *n* multiples of 5 not 3. We can move the cursor to highlight only the 3 node. Now *delete* the 3 node by typing ^D. The 3 node has been replaced with a meta node. A 5 may be entered just as the 3 was earlier.

Another way to do this is with the *replace* command, ^X^R. If the cursor is on the 3, the replace command prompts with *Replace by: 3*, the *delete* key removes the 3 and allows us to enter 5.

```
WHILE i++ LT n
    DO COLLECT 5 * i;

$stat_decl;
```

Another useful set of commands are *kill* (^K) and *yank* (^Y). ^K deletes the current highlighted subtree and save a copy of it in the *kill buffer*. ^Y then inserts the subtree from the *kill buffer* at the current meta node. These commands are useful for moving copies of program sections from one location to another. Note that the meta node where the ^Y is executed must be of the proper type to accept the root of the kill buffer. For example, if a *declaration* is deleted with ^K, a subsequent ^Y at an integer node does not work.

Searching

The editor has a string searching mechanism. The command ^S prompts for a string to search for. The search begins at the current highlighted node on the display and continues until the first occurrence of the string is found or until the end of the program is reached. ^R provides a reverse search from the current node to the top of the program. Searching with this structure editor is different from searching with an editor such as emacs. For instance, using the above example, searching for the string "WHILE" results in the entire subtree being highlighted. It is impossible to move the cursor to highlight just the five letters "WHILE" since it is just part of the syntactic sugar for the node.

Window Operations

The editor provides a number of commands for window manipulation. The *current* window contains the highlighted cursor. The *divide-window* command, ^X2, splits the current window into two windows. Only one of the two has the highlighted cursor in it, and this is the new current window. The cursor in the

current window can now be moved to a distant place in the tree while the other window remains in the original location. The *next-window* (^XN) and *previous-window* (^XP) commands move the current cursor between the two windows allowing alternate editing in distant places. ^X1 makes the current window the only window on the screen. ^XD deletes a named window (remember that each window has its name in the status line).

I.3. Run-Time Support

Evaluation

There are currently three editor commands that can be used to provide run-time support to the programmer.

1. **evaluate** - This command causes the subtree highlighted by the current cursor to be evaluated by the interpreter.
2. **Intonode** - This command takes as input a line number and moves the current cursor to the appropriate line.
3. **nodetoln** - This command returns the line number that corresponds to the position of the cursor.

The *evaluate* command is currently the editor's only way of sending program information to the interpreter. Subtrees that can be evaluated are *declarations*, *statements* or any kind of *expression*. Since AML/X is an *expression-oriented* language like LISP, this command is useful for examining many program fragments. Since the language itself provides for run-time support with debugging primitives, this evaluation routine is sufficient to provide unrestricted access to the run-time support offered by the interpreter.

The other two commands are intended to be used when debugging. Error messages from the interpreter will often include a reference to the line number at which the error was detected. The command, *Intonode*, facilitates locating the source of errors. When setting breakpoints, the AML/X debugging routines may require a linenumber as input. *Nodetoln* provides such line numbers.

In order to execute these commands, type *ESC-X <command name> <CR>*. When typing the command name it is only necessary to type enough of the name to uniquely identify it to the system. Typing the <space> bar will then complete the command name.

To illustrate how evaluation is done, consider the example from the last section.

```
WHILE i++ LT n
    DO COLLECT 5 * i;

$stat decl;
```

Move the cursor to highlight only the 5. Since an *integer* can be an *expression* in AML/X, we can evaluate it. Now type *ESC-X* and get the ":" prompt. Typing "ev" uniquely identifies the *evaluate* command to the system. Typing the <space> bar completes the command: ": evaluate". If you typed only "e" instead of "ev" the system would have displayed, in the help window, the names of other commands that could be executed and begin with "e". Typing <CR> now causes evaluation to occur. The result of the evaluation and the AML/X interpreter prompt are displayed in the evaluation pop up window.

```
AML:
5
```

If we move the user to highlight *5 * i* and try to evaluate it, we will get an error message.

```
WHILE i++ LT n
    DO COLLECT 5 * i;

$stat_decl;
```

The error message is displayed in the evaluation pop-up window. This message indicates the linenumber at which the error was detected. The *character* number indicates the number of the leaf in the line where the error was detected.

```
*** EXCEPTION EXCP_NOTNUM:
   Number (INT, LONG_INT, REAL, or LONG_REAL) expected. ***
    Operator: *
    Operands: < 5, UNBOUND >
    File:
    Line: 2
    Character: 2
    Data: UNBOUND
```

When executing *lntonode*, AMPE returns a prompt for input, "move cursor to line number: ". If we enter 2, the cursor is moved to the first leaf node in line number 2. Executing *nodetoln* at this point would also return 2. You can experiment with moving the cursor to other line numbers as well.

Debugging

The AML/X interpreter provides interactive debugging facilities. AML/X provides debugging commands that allow the programmer to:

1. **Stop Execution** - there are commands for setting breakpoints.
2. **Display and Assign** - once execution is stopped, any variable that is active or defined may be displayed or have its value changed.
3. **Trace Program Execution** - the execution of various statements may be traced during execution.
4. **Altering the Programming Environment** - the interpreter's environment can be altered without restarting the program.
5. **Create Debugging Scripts** - debugging scripts can be stored in a separate file that can be loaded when a particular application needs to be debugged.

The reader is referred to the AML/X User's Manual [3] manual for a more in depth discussion of debugging techniques in AML/X.

In order to use the AML/X debugging commands it is necessary to first load the definitions of the debugging routines. Because of current restrictions on the evaluation of AML/X language constructs, this cannot be done.

References

1. A. N. Habermann and D. Notkin, "Gandalf: Software Development Environments", IEEE Transactions on Software Engineering, vol. SE-12 No. 12, December 1986, pp. 1117-1127.

2. Charles W. Krueger, A. N. Habermann, "The GANDALF Editor Generator Reference Manuals" in "The GANDALF System Reference Manuals," Technical Report, Computer Science Department, Carnegie-Mellon University, 1986.

3. L. R. Nackman, M. A. Lavin, R. H. Taylor, W. C. Dietrich Jr., "AML/X User's Manual," IBM Research Report RA 175, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, (1986).